Multiple-Source Context-Free Path Querying in Terms of Linear Algebra

Arseniy Terekhov
simpletondl@yandex.ru
Saint Petersburg State University
St. Petersburg, Russia

Vlada Pogozhelskaya
pogozhelskaya@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Vadim Abzalov
vadim.i.abzalov@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Timur Zinnatulin
!!!@!!!
Saint Petersburg State University
St. Petersburg, Russia

Semyon Grigorev
s.v.grigoriev@spbu.ru
semyon.grigorev@jetbrains.com
Saint Petersburg State University
St. Petersburg, Russia
JetBrains Research
St. Petersburg, Russia

ABSTRACT

Context-Free Path Querying (CFPQ) allows one to use context-free grammars to express paths constraints in navigational graph queries. Algorithms for CFPQ studied actively for a long time, but no one graph database provide full-stack support of CFPQ. In this work we provide multiple-source version of Azimov's CFPQ algorithm, which, as shown by Arseniy Terekhov is applicable for real-world graph analysis. This step allows us to make the algorithm more practical and integrate it into RedisGraph graph database. In order to provide full-stack support we also implement Cypher graph query language extension that allows one to express context-free constraints. As a result, we provide the first, in our knowledge, full-stack support of CFPQ for graph database. Our evaluation shows that the provided solution is applicable for real-world graph analysis.

1 INTRODUCTION

Language-constrained path querying [3] is a way to find paths in edge-labeled graphs with constraints are formulated in terms of language which restrict words formed by paths: the word formed by path's labels concatenation should be in the specified language. This way is very natural for navigational queries in graph databases, and one of the most popular languages which are used for such constraints is a regular language. But in some cases, regular languages are not expressive enough, as a result, context-free languages gain popularity. Constraints in the form of context-free languages, or context-free path querying (CFPQ), can be used for RDF analysis [17], biological data analysis [15], static code analysis [14, 18], and in other areas.

Big amount of research done on CFPQ, a number of CFPQ algorithms were proposed, but the application of context-free constraints for real-world data analysis faced with some problems. The first problem is a bad performance of proposed algorithms on real-world data, as was shown by Jochem Kuijpers et al. [9]. The second problem is that there are no graph databases with full-stack support of CFPQ, the main effort was made in algorithms and their theoretical properties research. This fact hinders research of problems reducible to CFPQ, thus it hinders the development of new solutions for some problems. For example, recently graph segmentation in data provenance analysis was reduced to CFPQ [11], but authors faced the problem during the evaluation of the proposed approach: no one graph database support CFPQ.

In [2] Rustam Azimov proposed a matrix-based algorithm for CFPQ. This algorithm is one of promising way to solve the first problem and provide performant solution for real-world data analysis, as was shown by Nikita Mishim et al. in [12] and Arseniy Terekhov et al. in [16]. But this algorithm always computes information (reachability facts or single path which satisfies constraints) for all pairs of vertices in the graph, namely it solves *all-pairs* context-free path querying problem. Handling of all possible pairs is unreasonable in some real-world scenarios when one can provide a relatively small set of start vertices or even single start vertex.

While all-pairs context-free path querying is a classical problem that investigates in a number of works, there is no, in our knowledge, solutions for single-source and multiple-source CFPQ. In this work we propose a matrix-based *multiple-source* (and *single-source* as a partial case) CFPQ algorithm.

To solve the second problem, we provide full-stack support of CFPQ for the RedisGraph[1] [4] graph database. We implement a Cypher query language extension[2] that allows one to express context-free constraints, and extend the RedisGraph to support this extension. In our knowledge, it is the first full-stack implementation of CFPQ.

To summarize, we make the following contribution in this paper.

(1) We modify Azimov's matrix-based CFPQ algorithm and provide a multiple-source matrix-based CFPQ algorithm. As a partial case, it is possible to use our algorithm in a single-source scenario. Our modification still based on linear algebra, hence it is simple to implementation and allows one to use high-performance libraries and utilize modern parallel hardware for queries evaluation.
(2) We evaluate two version the proposed algorithm: with caching of results which should helps to reduce multiple

[1] RedisGraph graph database Web-page: https://redislabs.com/redis-enterprise/redis-graph/. Access date: 19.07.2020.
[2] Proposal which describes path patterns specification syntax for Cypher query language: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc. The proposed syntax allows one to specify context-free constraints. Access date: 19.07.2020.

calculation of the same data, and without caching (naïve). Our evaluation shows that the naïve version is performant than the version with results caching in almost the all cases. Moreover, this version is more memory-efficient. Thus it is good choice for implementation in real-world graph database.

(3) We provide full-stack support of CFPQ by extending the RedisGraph graph database. To do it, we extend Cypher with syntax allows one to express context-free constraints, implement the proposed algorithm in a RedisGraph backend, and support new syntax in the RedisGraph query execution engine. Finally, we evaluate the proposed solution and show that it is performant and memory-efficient enough to be applicable for real-world graph querying.

2 PRELIMINARIES

In this section we introduce common definitions in graph theory and formal language theory which will be used in this paper. Also, we provide brief description of Azimov's algorithm which is used as a base of our solution.

2.1 Basic definitions of Graph Theory

In this work we use labeled directed graph as a data model and define it as follows.

Definition 2.1. Labeled directed graph is a tuple of six elements $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$, where

- Σ_V and Σ_E is a set of labels of vertices and edges respectively, such that $\Sigma_V \cap \Sigma_E = \varnothing$.
- V is a set of vertices. For simplicity, we assume that the vertices are natural numbers from 0 to $|V| - 1$.
- $E \subseteq V \times V$ is a set of edges.
- $\lambda_V : V \to 2^{\Sigma_V}$ is a function that maps a vertex to a set of its labels, which can be empty.
- $\lambda_E : E \to 2^{\Sigma_E} \setminus \{\varnothing\}$ is a function that maps a edge to a not empty set of its labels, so each edge must have at least one label.

□

Labeled graph is a part of widely-used *property graph* data model [1] and allows one to use in navigation queries not only edge labels but also vertex labels.

An example of the labeled directed graph D_1 is presented in figure 1. Here the sets of labels $\Sigma_V = \{x, y\}$ and $\Sigma_E = \{a, b, c, d\}$. We omit vertex labels set if it is empty.

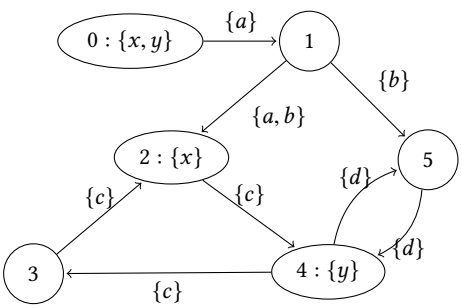

Figure 1: The example of input graph D_1

Definition 2.2. Path π in the graph $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ is a finite sequence of vertices and edges $(v_0, e_0, v_1, e_1, ..., e_{n-1}, v_n)$, where $\forall i \mid 0 \le n\ v_i \in V, \forall j \mid 1 \le j \le n\ e_j = (v_j, v_{j+1}) \in E$.

We denote the set of all possible paths in the graph D as $\pi(D)$. □

Definition 2.3. An *adjacency matrix* M of the graph D is a square $|V| \times |V|$ matrix, such that

$$M[i,j] = \begin{cases} \lambda_E((i,j)), & (i,j) \in E \\ \varnothing, & else \end{cases}$$

□

Adjacency matrix M of the graph D_1 (fig. 1) is

$$M = \begin{pmatrix} \varnothing & \{a\} & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{a,b\} & \varnothing & & \{b\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{d\} & \varnothing \\ \varnothing & \varnothing & \{c\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{c\} & \varnothing & \{d\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{d\} & \varnothing \end{pmatrix}.$$

Definition 2.4. Let M be an adjacency matrix of the graph D. Then the *adjacency matrix of label* $l \in \Sigma_E$ of graph D is a $|V| \times |V|$ matrix $\mathcal{E}^l$, such that

$$\mathcal{E}^l[i,j] = \begin{cases} 1, & l \in M[i,j] \\ 0, & else \end{cases}$$

□

Definition 2.5. Boolean decomposition of adjacency matrix M of the graph D is a set of Boolean matrices

$$\mathcal{E} = \{\mathcal{E}^l \mid l \in \Sigma\},$$

where $\mathcal{E}^l$ is the adjacency matrix of label l. □

For example, adjacency matrix M of the example graph D_1 can be represented as a set of four Boolean matrices $\mathcal{E}^a$, $\mathcal{E}^b$, $\mathcal{E}^c$ and $\mathcal{E}^d$ such that

$$\mathcal{E}^a = \begin{pmatrix} . & . & . & . & . & . \\ 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \end{pmatrix}, \mathcal{E}^b = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & 1 & . & . & . & . \end{pmatrix},$$

$$\mathcal{E}^c = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \\ . & . & 1 & . & . & . \\ . & . & . & . & . & . \end{pmatrix}, \mathcal{E}^d = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & 1 \\ . & . & . & . & 1 & . \end{pmatrix}.$$

Definition 2.6. An *vertices label matrix* H of the graph D is a square $|V| \times |V|$ matrix, such that

$$H[i,j] = \begin{cases} \lambda_V(i), & i = j \\ \varnothing, & else \end{cases}$$

□

The vertices label matrix H of the example graph D_1 is

$$H = \begin{pmatrix} \{x,y\} & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{x\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{y\} & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \end{pmatrix}.$$

Definition 2.7. Let H be a vertices label matrix of graph D. Then the *vertices matrix of label* l is a square $|V| \times |V|$ matrix $\mathcal{V}^l$, such that

$$\mathcal{V}^l[i,j] = \begin{cases} 1, & l \in H[i,j] \\ 0, & else \end{cases}$$

Definition 2.8. Boolean decomposition of vertices label matrix H of the graph D is the set of Boolean matrices

$$\mathcal{V} = \{V^l \mid l \in \Sigma\},$$

where $\mathcal{V}^l$ is a vertices matrix of label l.

Vertices label matrix H of the graph D_1 can be decomposed into a set of the following Boolean matrices:

$$\mathcal{V}^x = \begin{pmatrix} 1 & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \end{pmatrix}, \mathcal{V}^y = \begin{pmatrix} 1 & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & 1 & . \\ . & . & . & . & . & . \end{pmatrix},$$

2.2 Basic Definitions of Formal Languages

We use context-free grammars as paths constraints. Thus we should define context-free languages and grammars.

In other worlds concatenation of two sets contains all concatenations of elements from the first set with all elements from the second one.

Definition 2.9. Context-free grammar is a tuple $\mathcal{G} = (N, \Sigma, P, S)$, where

- N is a finite set of nonterminals
- Σ is a finite set of terminals
- P is a finite set of productions of the following forms: $A \to \alpha,\ A \in N,\ \alpha \in (N \cup \Sigma)^*$
- S is a start nonterminal

□

Definition 2.10. Context-free language is a language generated by a context-free grammar $\mathcal{G}$:

$$L(\mathcal{G}) = \{w \in \Sigma^* \mid S \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} w\}$$

Where $S \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} w$ denotes that a string w can be generated from a starting non-terminal S using some sequence of production rules from P. □

Definition 2.11. Context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$ is said to be in *Chomsky normal form* if all productions in P are in one of the following forms:

- $A \to BC,\ A \in N,\ B, C \in N \setminus S$
- $A \to a,\ A \in N,\ a \in \Sigma$
- $S \to \varepsilon, where \varepsilon$ is an identity element of Σ^*, or an empty string.

□

Definition 2.12. Context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$ is said to be in *weak Chomsky normal form* if all productions in P are in one of the following forms:

- $A \to BC,\ A, B, C \in N$
- $A \to a,\ A \in N,\ a \in \Sigma$
- $A \to \varepsilon,\ A \in N$

□

In other words, weak Chomsky normal form differs from Chomsky normal form in the following:

- ε can be derived from any non-terminal;
- S can be at a right part of productions.

Since matrix-based CFPQ algorithms processes grammars only in weak Chomsky normal form, it should be noted that every context-free grammar can be transformed into an equivalent one in this form.

Consider the following example of the context-free grammar $G_1 = (N, \Sigma, P, S), where\ N = \{S\},\ \Sigma = \{c, d, y\}$, and P contains two rules:

$$S \to c\ S\ d$$
$$S \to c\ y\ d.$$

This grammar generates the context-free language

$$L(G_1) = \{c^n y d^n, n \in \mathbb{N}\}.$$

One can get the following grammar as a result of the transformation of the G_1 to weak Chomsky normal form:

$$\begin{aligned} S &\to C\ E & C &\to c \\ S &\to C\ S_1 & Y &\to y \\ E &\to Y\ D & D &\to d \\ S_1 &\to S\ D \end{aligned}$$

2.3 Context-Free Path Querying

Definition 2.13. Let $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ be a labeled graph, $G = (N, \Sigma_V \cup \Sigma_E, P, S)$ be a context free grammar. Then a *context free relation* with grammar G on the labeled graph D is the following relation $R_{G,D} \subseteq V \times V$:

$$R_{G,D} = \{(v, to) \in V \times V \mid \exists \pi = (v_1, e_1, v_2, e_2, \ldots, e_n, v_n) \in \pi(D) : v_1 = v, v_n = to,\ l(\pi) \cap L(G) \neq \varnothing\},$$

where $l(\pi) \subset (\Sigma_V \cup \Sigma_E)^*$ is the set of possible labels along the path π:

$$l(\pi) = \lambda_V(v_1)^* \cdot \lambda_E(e_1) \cdot \lambda_V(v_2)^* \cdot \lambda_E(e_2) \cdot \ldots \cdot \lambda_E(e_n) \cdot \lambda_V(v_n)^*$$

□

For example, in the labeled graph presented in figure 1 there is the path

$$\pi = 3 \xrightarrow{\{c\}} 5 : \{y\} \xrightarrow{\{d\}} 6$$

from the vertex 3 to vertex 6. Labels along this path form the sequence cyd. It can be observed that this sequence satisfies context-free constraints of the above grammar G_1:

$$S \Rightarrow CE \Rightarrow cE \Rightarrow cYD \Rightarrow cyD \Rightarrow cyd$$

Hence $l(\pi) \cap L(G_1) \neq \varnothing$ and the pair $(3, 6) \in R_{G_1,D}$.

Note that the proposed definition, namely zero or more repetition of each vertex label allows one freely omit labels or use them in arbitrary order in case when there are more then one label for vertex. On the other hand, it makes valid queries that use one label more than ones. In some cases such behavior may looks strange, but it depends on semantics on query language, so we argue that it is necessary to formalize semantics of graph query language first, which is task for the future.

Finally, we are can define context-free path querying problem as follows.

Definition 2.14. Context-free path querying problem is the problem of finding context-free relation $R_{G,D}$ for a given directed labeled graph D and a given context-free grammar G. □

In other words, the result of context-free path query evaluation is a set of vertex pairs such that there is a path between them and this path forms a word from the given language.

For graph D_1 and context-free free grammar G_1 the relation

$$R_{G_1,D_1} = \{(2,4),(2,5),(3,4),(3,5),(4,4),(4,5)\}.$$

Note that any relation $R_{G,D}$ can be represented as a Boolean matrix:

$$T[i,j] = 1 \iff (i,j) \in R_{G,D}.$$

In our example, R_{G_1,D_1} can be represented as follows:

$$T = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & 1 & 1 \\ . & . & . & . & 1 & 1 \\ . & . & . & . & 1 & 1 \\ . & . & . & . & . & . \end{pmatrix}.$$

In case when one restricts a set of start vertices we can say about *multiple-source context-free path querying*.

Definition 2.15. Suppose Src is a given set of start vertices, then *multiple-source context-free path querying problem* for the given Src is the problem of finding context-free relation

$$R^{Src}_{G,D} \subseteq Src \times V \subseteq R_{G,D}$$

for a given directed labeled graph D and a given context-free grammar G. Namely, we restrict start vertices of the paths of interest to be a vertices from the given set □

As a partial case, one can get a single-source version of CFPQ by restrict Src to be a set of one element. If in the previous example we set $Src = \{2\}$, then the result is

$$R^{\{2\}}_{G_1,D_1} = \{(2,4),(2,5)\}.$$

For unification we can represent the $R^{\{2\}}_{G_1,D_1}$ as a Boolean matrix:

$$T = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & 1 & 1 \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \end{pmatrix}.$$

2.4 Matrix-Based Algorithm

Our algorithm is based on the Azimov's CFPQ algorithm [2] which is based on matrix operations. This algorithm allows one to use high-performance linear algebra libraries and utilize modern parallel hardware for CFPQ.

Let $G = (N, \Sigma, P, S)$ be the input grammar, the input edge-labeled graph $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ and language L over alphabet Σ. The matrix-based algorithm for CFPQ can be expressed in terms of operations over Boolean matrices as showed in listing 1. This fact simplifies implementation of the algorithm.

Note, that the provided algorithm returns not only context-free relation $R_{G,D}$ but a set of context-free relations $R_{A,D} \subseteq V \times V$ for every $A \in N$, thus it provides information about paths which worm words derivable from any nonterminal in the given grammar. Also, this algorithm handles only edge labels.

Algorithm 1 Context-free path querying algorithm

```
1: function evalCFPQ(D = (V, E, Σ_V, Σ_E, λ_V, λ_E), G = (N, Σ, P, S))
2:     n ← |V|
3:     T ← {T^{A_i} | A_i ∈ N, T^{A_i} is a matrix n × n, T^{A_i}_{k,l} ← false}
4:     for all (i, j) ∈ E, A_k | λ_E(i, j) = x, A_k → x ∈ P do T^{A_k}_{i,j} ← true
5:     for all A_k | A_k → ε ∈ P do
6:         for all i ∈ {0, ..., n − 1} do T^{A_k}_{i,i} ← true
7:     while any matrix in T is changing do
8:         for A_i → A_j A_k ∈ P do T^{A_i} ← T^{A_i} + (T^{A_j} × T^{A_k})
9:     return T
```

As was shown by Nikita Mishin et al. [12] and Arseniy Terekhov et al. [16], this algorithm can be implemented using various high-performance programming techniques (including GPGPU utilization), and it is applicable for real-world graph analysis. But this algorithm solves *all-pairs* version of CFPQ: it finds all pairs of vertices in the given graph, such that there exist a paths between them which forms a word in the given language. Thus it is impractical in cases when we need only paths which start from specific set of vertices, especially if this set is relatively small. Moreover, Azimov's algorithm operates over adjacency matrices of full graphs, as a result it requires a huge amount of memory, which may be a problem for real-world graph database.

3 MATRIX-BASED MULTIPLE-SOURCE CFPQ ALGORITHM

In this section we introduce two versions of multiple-source matrix-based CFPQ algorithm. This algorithm is a modification of Azimov's matrix-based algorithm for CFPQ and its idea is that we cut off those vertices from which we are not interested in paths.

In order to simplify Azimov's algorithm modification and the final algorithm description, we simplify the input graph to have only edge labels. Note, that we always can convert the original graph into such form. To do it we should add loops into vertices in the following way: for the vertex i we add an edge $i \xrightarrow{x} i$ iff $\lambda_V(i) = x$ and $x \neq \varnothing$. This way we can switch to edge-labeled graph with the same number of vertices with preserving of the defined semantics of CFPQ.

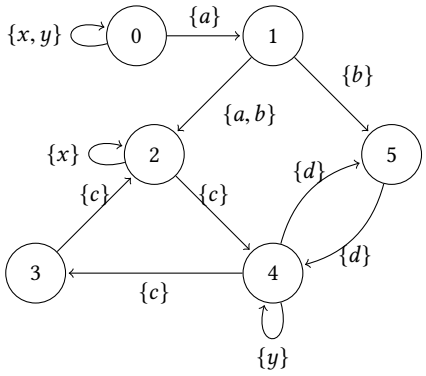

Figure 2: The example of D'_1: the modified input graph D_1

The adjacency matrix M of the graph D_1' is

$$M = \begin{pmatrix} \{x,y\} & \{a\} & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{a,b\} & \varnothing & & \{b\} \\ \varnothing & \varnothing & \{x\} & \varnothing & \{d\} & \varnothing \\ \varnothing & \varnothing & \{c\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{c\} & \{y\} & \{d\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{d\} & \varnothing \end{pmatrix}.$$

Note that this transformation is impractical for real-world graphs, thus we use it only for algorithm description.

The first version of multiple-source algorithm is the Azimov's algorithm equipped with vertices filtering. Let $G = (N, \Sigma, P, S)$ be the input context-free grammar, $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ be the input graph and Src be the input set of start vertices. The result of the algorithm is a Boolean matrix which represents relation $R^{Src}_{S,D}$.

Algorithm 2 Multiple-source context-free path querying algorithm

```
1:  function MultiSrcCFPQ(D = (V, E, Σ_V, Σ_E, λ_V, λ_E), G = (N, Σ, P, S), Src)
2:      T ← {T^A | A ∈ N, T^A ← ∅}        ▷ Matrix in which every element is ∅
3:      TSrc ← {TSrc^A | A ∈ N \ S, TSrc^A ← ∅}   ▷ Matrix for input vertices in which every element is ∅
4:      for all v ∈ Src do                 ▷ Input matrix initialization
5:          TSrc^S_{v,v} ← true
6:      for all A → x ∈ P do               ▷ Simple rules initialization
7:          for all (v, to) ∈ E, λ_E(v, to) = x do
8:              T^A_{v,to} ← true
9:      while T or TSrc is changing do     ▷ Algorithm's body
10:         for all A → BC ∈ P do
11:             M ← TSrc^A * T^B
12:             T^A ← T^A + M * T^C
13:             TSrc^B ← TSrc^B + TSrc^A
14:             TSrc^C ← TSrc^C + getDst(M)
15:     return T^S
16:
17: function getDst(M)
18:     A ← ∅
19:     for all (v, to) ∈ V^2 | M_{v,to} = true do
20:         A_{to,to} ← true
21:     return A
```

In order to solve the single-source and multiple-source CFPQ problem Azimov's algorithm was modified: each time, when we apply grammar rule (Boolean matrix multiplication $T_A = T_A + T_B \cdot T_C$ for each $A \to BC \in P$ represented in line **8** of Algorithm 1) we should save only vertices of interest. To do it, matrix multiplication was supplemented with one more matrix multiplication $T_A = T_A + (TSrc^A \cdot T_B) \cdot T_C$, where $TSrc^A$ — matrix of vertices to calculate the paths from (lines **11-13** of the Algorithm 2). Also, after every iteration of while loop this is necessary to update the set of vertices paths from which we need to calculate. To do this, the function **getDst**, represented in lines **17-21**, is called at line **14**. Thus, the modified algorithm supports the frontier of the actual vertices and updates it on each iteration. As a result it does not calculate the paths from all vertices in case of query to calculate the paths small set of vertices.

========== We proposed the variant of the algorithm that can calculate the paths from a certain set of vertices, however there are such scenarios when queries are partially or completely repeated. In such cases it would be useful to add data caching to improve the performance. The problem is that every time we want to find all paths from the certain set of vertices, the Algorithm 2 calculates everything from scratch. Since recalculating might take the significant amount of time, we modified multiple-source CFPQ algorithm to specify it for such scenarios. This version stores all the vertices the paths from which have already been calculated in cash *index*, which is used to filter such vertices in line **3** of Algorithm 3. Thus, modified algorithm calculates paths from the particular vertex only once.

Algorithm 3 Optimized multiple-source context-free path querying algorithm

```
1:  function MultiSrcCFPQSmart(index = (D, G, T, TSrc), Src)
2:      TNewSrc ← {TNewSrc^A | A ∈ N \ S, TNewSrc^A ← ∅}
3:      for all v ∈ Src | index.TSrc_{v,v} = false do
4:          TNewSrc^S_{v,v} ← true
5:      while index.T or TNewSrc is changing do
6:          for all A → BC ∈ P do
7:              M ← TNewSrc^A * index.T^B
8:              index.T^A ← index.T^A + M * index.T^C
9:              TNewSrc^B ← TNewSrc^B + TNewSrc^A \ index.TSrc^B
10:             TNewSrc^C ← TNewSrc^C + getDst(M) \ index.TSrc^C
```

3.1 Implementation Details

All of the above versions have been implemented[3] using GraphBLAS framework that allows you to represent graphs as matrices and work with them in terms of linear algebra. For convenience, all the code is written in Python using pygraphblas[4], which is Python wrapper around GraphBLAS API and based on SuiteSparse:GraphBLAS[5] [5] — the full implementation of GraphBLAS standard. This library is specialized for working with sparse matrices, which most often appear in real graphs. Also, it should be noted that, despite the fact that the function **getDst** does not seem to be expressed in terms of linear algebra, the implementation used the function **reduce_vector** from pygraphblas that reduces matrix to a vector, with which further work takes place.

3.2 Algorithm Evaluation

We evaluate both described version of multiple-source algorithm on real-world graphs. For evaluation, we use a PC with Ubuntu 20.04 installed. It has Intel core i7-4790 CPU, 3.60GHz, and DDR3 32Gb RAM. As far as we evaluate only algorithm execution time, we store each graph fully in RAM as its adjacency matrix in sparse format. Note, that graph loading time is not included in the result time of evaluation.

[3] GitHub repository with implemented algorithms: https://github.com/JetBrains-Research/CFPQ_PyAlgo, last accessed 28.08.2020

[4] GitHub repository of PyGraphBLAS library: https://github.com/michelp/pygraphblas

[5] GitHub repository of SuiteSparse:GraphBLAS library: https://github.com/DrTimothyAldenDavis/SuiteSparse

Table 1: Graphs for CFPQ evaluation

Graph	#V	#E	#subClassOf	#type	#broaderTransitive
core	1323	3636	178	706	0
pathways	6238	18 598	3117	3118	0
gohierarchy	45 007	980 218	490 109	0	0
enzyme	48 815	117 851	8163	14 989	8156
eclass_514en	239 111	523 727	90 962	72 517	0
geospecies	450 609	2 311 461	0	89 062	20 867
go	582 929	1 758 432	94 514	226 481	0

For evaluation we use graphs and queries from CFPQ_Data dataset[6] Detailed information, such as number if vertices and edges, and number of edges with specific label, on graphs which we select for evaluation is provided in table 1. We use classical same-generation queries g_1 (eq. 1) and g_2 (eq. 2) which are used in other works for CFPQ evaluation. Also we use *geo* (eq. 3) query which was provided by J. Kuijpers et. al [9] for *geospecies* RDF. Note that in queries we use $\overline{x}$ notation to denote inverse of x relation and respective edge.

$$S \rightarrow \overline{subClassOf}\ S\ subClasOf \mid \overline{type}\ S\ type \mid \overline{subClassOf}\ subClasOf \mid \overline{type}\ type \tag{1}$$

$$S \rightarrow \overline{subClassOf}\ S\ subClasOf \mid subClassOf \tag{2}$$

$$S \rightarrow broaderTransitive\ S\ \overline{broaderTransitive} \mid broaderTransitive\ \overline{broaderTransitive} \tag{3}$$

Our main goal is to compare behavior of two proposed versions of the algorithm. To do it we measure query execution time for both versions for different sizes of star vertex set. Namely, for each graph we split all vertices into disjoint subsets of fixed size. After that, for each subset we evaluate queries using the given subset as a set of start vertices.

For each graph we evaluate all three queries. Results of evaluation is presented in figures 3–9. We use standard violin plot with median to show distribution of results, time is measured in seconds. For number of input graphs we provide additional figures for small chunks in order to analyze these cases carefully: figures 10–12.

First of all, we can see, that even for cases when graph does not contain edges which are used in query, chunk processing time grows with size of chunk. For example, look at the results for *geo* query for all graphs except *geospecies* and *enzyme*. Thus, preliminary check of existence of edges of interest may be useful in some cases.

Also, we can see, that chunk processing time significantly depends on graph structure. For example, for chunks of size 10000 and query g_1, *go* graph querying requires more than 5 seconds (fig. 4), while *geospecies* graph querying requires less than 0.5 seconds (fig. 6).

Comparison of two version of algorithm shows that algorithm without caching is significantly faster in almost all cases, even when graph does not contain edges of interest. Analysis of results for small chunks (fig. 10–12) shows that it is always true. For example, for *eclass_514en* graph and query g_2 (fig. 11) median time for algorithm with caching is slightly better than for the naïve version. On the other hand, for *geospecies* graph and *geo* query (fig.11) algorithm with caching is drastically slower that the naïve version. At the same time, for *go* graph and g_2 query median time for both versions are comparable, while time for worst queries is better for the naïve version. Moreover, caching requires additional memory in comparison with naïve version of the algorithm. Thus we can conclude, that query results caching

[6]CFPQ_Data is a dataset for CFPQ evaluation which contains both synthetic and real-world data and queries https://github.com/JetBrains-Research/CFPQ_Data, last accessed 28.08.2020.

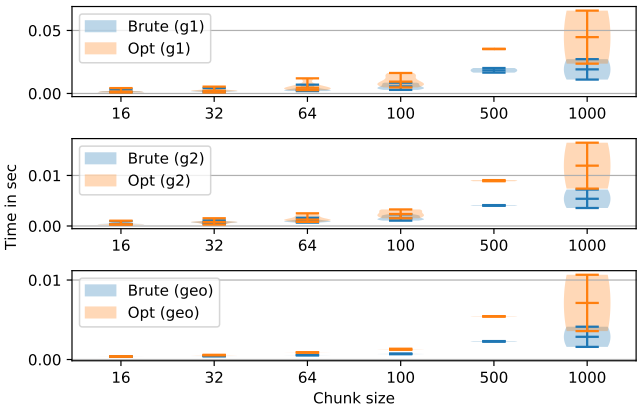

Figure 3: Performance of *core* graph querying

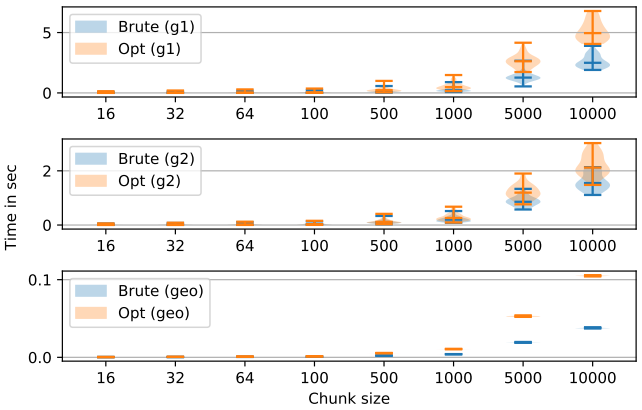

Figure 4: Performance of *go* graph querying

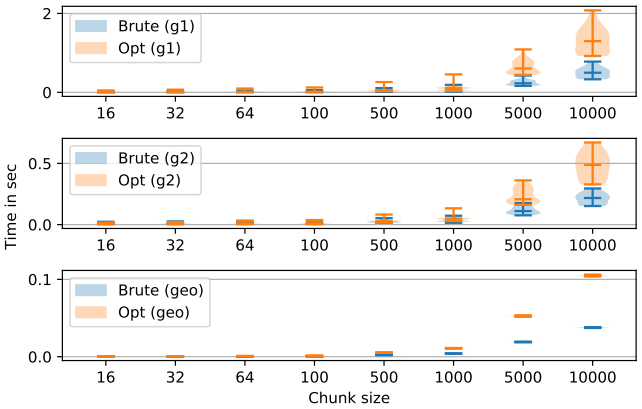

Figure 5: Performance of *eclass_514en* graph querying

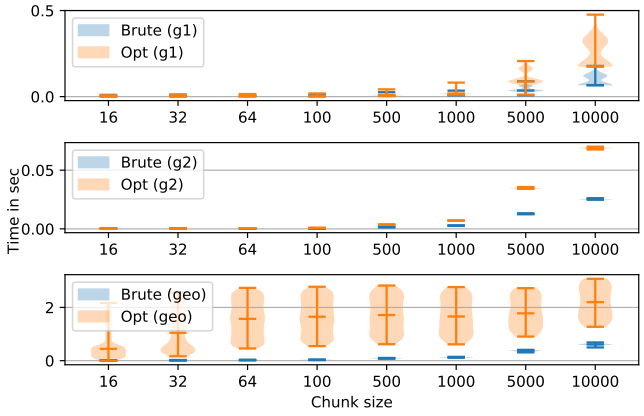

Figure 6: Performance of *geospecies* graph querying

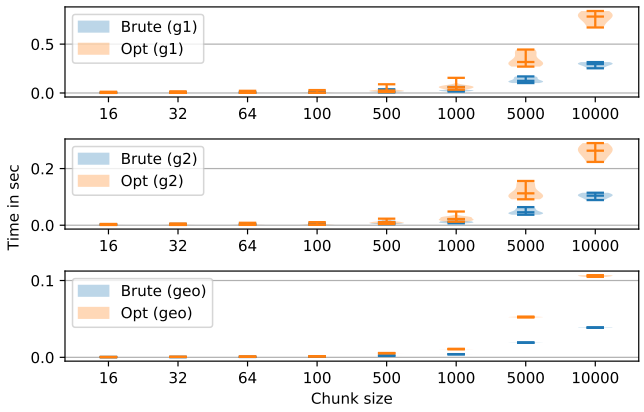

Figure 7: Performance of *enzyme* graph querying

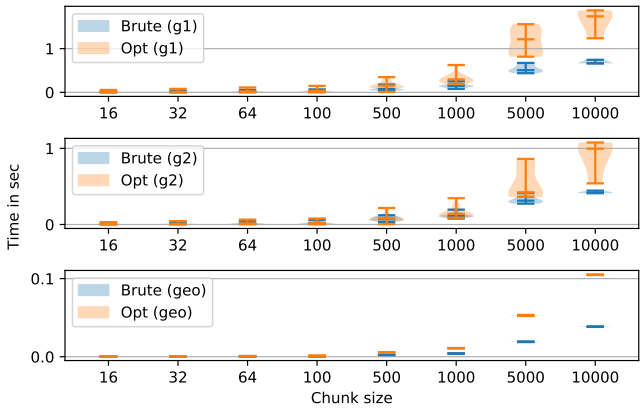

Figure 8: Performance of *gohierarchy* graph querying

introduces significant overhead and does not lead to significant performance improvements. Also we can conclude that small chunk processing using the naïve version is fast enough: the worst time in our experiments is about 0.2 seconds (fig. 12, query G_1).

As a result, we can conclude that caching is not useful for multiple-source CFPQ for evaluated cases even if one want to

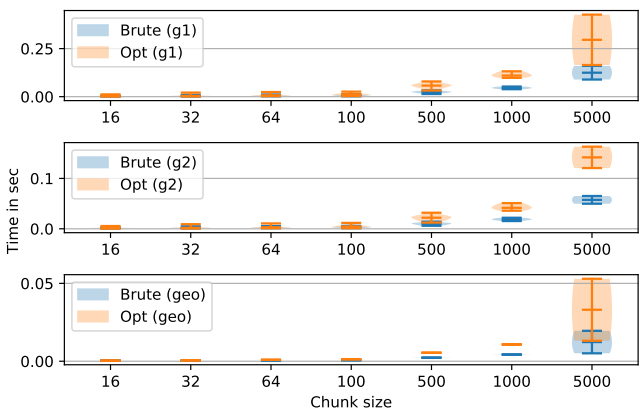

Figure 9: Performance of *pathways* graph querying

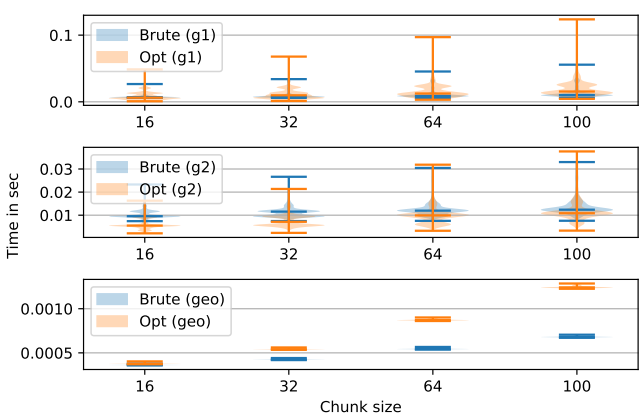

Figure 10: Performance of *eclass_514en* graph querying with small chunks

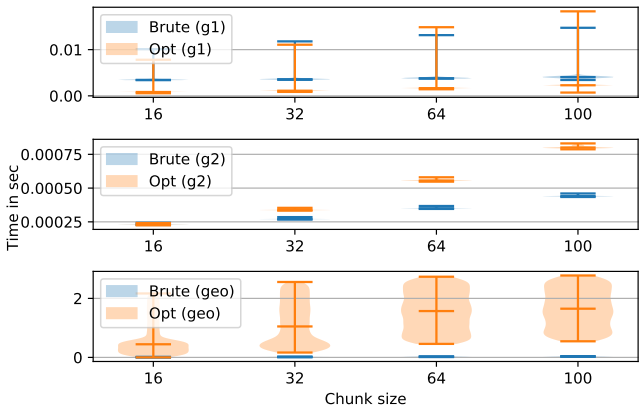

Figure 11: Performance of *geospecies* graph querying with small chunks

process several chunks sequentially, or even process full graph chunk-by-chunk. Thus, we think that the naïve version of the algorithm is better for implementation in real-world graph database.

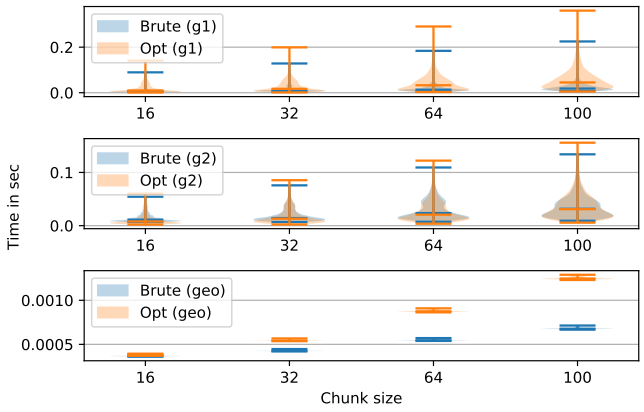

Figure 12: Performance of *go* graph querying with small chunks

4 CFPQ FULL-STACK SUPPORT

In order to provide full-stack support of CFPQ it is necessatry to choose an appropriatr graph database. It was shown by Arseniy Terekhov et al. in [16] that matrix-based algorithm can be naturally integrated into RedisGraph graph database because both, the algorithm and the database, operates over matrix representation of graphs. Moreover, RedisGraph supports Cypher as a query language and there is a proposal which describes Cypher extension which allows one to specify context-free constraints. Thus we choose RedisGraph as a base for our solution.

4.1 Cypher Extending

The first what we should do is to extend Cypher parser to be able to express context-free constraints. There is a description of the respective Cypher syntax extension[7], proposed by Tobias Lindaaker, but this syntax does not implement yet in Cypher parsers.

This extension introduces path patterns, which are powerful alternative to the original Cypher relationship patterns. Path patterns allow one to express regular constrains over basic patterns such as relationship and node patterns. Just like relationship patterns they can be specified in the `MATCH` clause between the node patterns.

Listing 4 Example of using a simple path pattern

```
1: MATCH (v)-/ [:a (:x) :b] | [:c (:y) :d] /->(to)
2: RETURN v, to
```

An example of query in extended syntax with a simple path pattern is provided in listing 4. In this example there are relationship patterns :a, :b, :c :d and node patterns (:x), (:y). The square brackets are used for grouping parts of the pattern. The | symbol denotes alternative between corresponding paths and the blank denotes sequence of paths. So the result of executing the query on the labeled graph D will be the following set of vertex pairs:

$$\{(v, to) : \exists \pi = (v, r_1, u, r_2, to) \in \pi(D) : \begin{bmatrix} a \in \lambda_E(r_1), x \in \lambda_V(u), b \in \lambda_E(r_2) \\ c \in \lambda_E(r_1), y \in \lambda_V(u), d \in \lambda_E(r_2) \end{bmatrix} \}$$

Main feature which allows one to specify context-free constraints is a *named path patterns*: one can specify a name for path pattern and after that use this name in other patterns, or in the same pattern. Using this feature, structure of query is pretty similar to context-free grammar in the Extended Backus-Naur Form (EBNF) [**?**].

Listing 5 Example of named path pattern

```
1: PATH PATTERN S = ()-/ [:a ~S :b] | [:a :b] /->()
2: MATCH (v)-/ ~S /->(to)
3: RETURN v, to
```

An example of named path patterns is presented in listing 5. Named patterns can be defined in the `PATH PATTERN` clause and referenced within any other path pattern. In order to explain the semantics of the query, consider contest-free grammar $G = (N, \Sigma, P, S)$ with $N = \{S\}, \Sigma = \{a, b\}$ and $P = \{S \to ab, S \to aSb\}$. Then $L(G) = \{a^n b^n : n \in \mathbb{N}\}$ specifies restrictions on the path labels and query result on the graph D will be the context free relation $R_{G,D}$(What you want to say? In Russian. [gsv]).

Thus this Cypher extension allows one express more complex queries including context-free path queries. RedisGraph database supports subset of Cypher language and uses `libcypher-parser`[8] library to parse queries. We extend this library by introducing new syntax proposed [7]. We implement[9] full extension, not only part which is necessary for simple CFPQ.

4.2 RedisGraph Intro (TODO: move to introduction)

Named path patterns described in subsection 4.1 allows one to specify context-free constrains on the paths. In order to support the execution of these types of queries we need to extend backend of the RedisGraph database and integrate a suitable CFPQ algorithm into it.

There are quite a few algorithms that solve CFPQ problem **??**, but their running time makes them unsuitable for practical use **??**. Recent studies **??** have shown that one can achieve high performance through the use of matrix-based algorithms. These studies were conducted to analyze the performance of the Rustam Azimov algorithm described in **??** and have shown that it is acceptable for practical application.

Using the Rustam Azimov algorithm one can only find paths between all pairs of vertexes at once and in some cases it is quite wasteful. Queries to graph databases can be specified so that when they are executed, it is required to find paths from a given set of initial vertices. This set can be quite small due to the different filtering specified in the query. For example in the listing 7 path pattern `-/ ~S /->` follows pattern `(v)-[r]->(u)` . The WHERE clause specifies some arbitrary predicate `p(v, r, u)` which also fixes a set of initial vertexes for a paths that must

[7]Formal syntax specification: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc#11-syntax. Access date: 19.07.2020.

[8]The `libcypher-parser` is an open-source parser library for Cypher query language. GitHub repository of the project: https://github.com/cleishm/libcypher-parser. Access date: 19.07.2020.

[9]The modified libsypher-pareser library with support of syntax for path patterns: https://github.com/YaccConstructor/libcypher-parser. Access date: 19.07.2020.

satisfy path pattern S. Depending on this predicate, this set of vertexes can have different sizes and for proper practical use the running time of the CFPQ algorithm should be sensitive to this.

Listing 6 ...

```
1: PATH PATTERN S = ()-/ :A [~S | ()] :B /->()
2: MATCH (v)-[r]->(u)-/ ~S /->(to)
3: WHERE p(v, r, u)
4: RETURN to
```

The Multi-Source algorithm described in **??** is sensitive to the initial set of vertices and is therefore well suited for graph database query scenarios. In addition, it is based on matrix operations and works with graphs as sparse matrices, so it is suitable for integration in RedisGraph.

4.3 RedisGraph extension

This section describes the implementation of support for executing queries with the extended syntax in the RedisGraph. Throughout this section, we consider executing the example query from listing 7 for the graph D_1 from Figure 1. $\mathcal{E}$ and $\mathcal{V}$ denotes boolean decompositions of adjacency and vertex label matrices of D_1 respectively.

Listing 7 Query with path patterns example

```
1: PATH PATTERN S = ()-/ [:c ~S :d] | [:c (:y) :d] /->()
2: MATCH (v:x)-[:a | :c]->()-/ :b ~S /->(to)
3: RETURN v, to
```

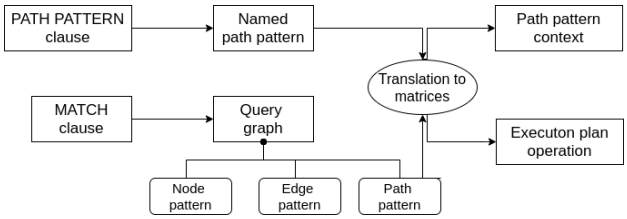

Figure 13: Extension diagram for building a query execution plan

4.3.1 Execution plan building. In the RedisGraph the main part of processing a query is building its execution plan. Execution plan consists of operations that perform basic processing such as filtering, pattern matching, aggregation and result construction. The diagram of its construction is shown in Figure 13. It can be divided into two parts —processing named and unnamed path patterns, which are described below(Improve it! [gsv]).

Let's consider the part that associated with unnamed path patterns. Unnamed path patterns relates to the pattern matching operations and is very similar to relationship patterns from the original Cypher. All pattern matching operations are derived from the MATCH clause that consists of relationship patterns and node patterns. In the example query there is a relationship pattern $r = -[:a \mid c]->$, path pattern $p = -/ :b \sim S /->$ and node pattern $n = (:x)$. In the first stage of processing, these patterns turn into an intermediate representation — the *query graph*. The nodes and edges of the query graph corresponds to node and relationship patterns. We extended query graph to be able to contain path patterns. Thus the query graph edges can be either relationship or path patterns, which are stored in a more convenient intermediate representation other than AST. The query graph for patterns p, r and n is shown in Figure 14.

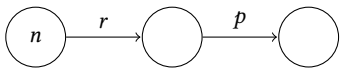

Figure 14: The example of input graph $\mathcal{G}$

At the second stage, the query graph is translated into algebraic expressions over matrices. The abstract syntax of an algebraic expression is provided in Figure 15. Thus the algebraic expression is an expression with addition, multiplication and transposition operations whose operands are matrices. To support references to named paths patterns in algebraic expressions we added a matrix operand *Ref(ref)* that stores a reference. In order to translate the query graph RedisGraph first linearizes it and then splits it into small paths. To support path patterns we extended the split processing so that each path pattern corresponds to exactly one path after query graph splitting. For example the query graph in Figure 14 is very simple and is divided into three patterns n, r and p. After that, each path is translated into a single algebraic expression. We developed the semantics of path patterns in terms of algebraic expressions and implemented translation. For example, node pattern n translates to $AlgExp(n) = \mathcal{V}^x$, relationship pattern r to $AlgExp(r) = \mathcal{E}^a + \mathcal{E}^c$ and path pattern p to $AlgExp(p) = \mathcal{E}^b * Ref(S)$.

Figure 15: Algebraic expression abstract syntax

$$
\begin{aligned}
AlgExpr = {} & (AlgExpr + AlgExpr) \mid \\
& (AlgExpr * AlgExpr) \mid \\
& Transpose(AlgExpr) \mid \\
& Matrix \mid \\
& Ref(ref)
\end{aligned}
$$

After obtaining algebraic expressions they are used to construct execution plan operations. Each operation is derived from a single algebraic expression that is involved in the further execution of the corresponding operation. For example for the $AlgExp(r)$ and $AlgExp(n)$ will be created *CondTraverse(AlgExp(r))* and *LabelScan(AlgExp(n))* operations respectively which already existed in RedisGraph. For expressions that correspond to path patterns we created a new *CFPQTraverse* operation. Thus algebraic expression of pattern p will be stored in the new *CFPQTraverse(AlgExp(p))* operation. During the query execution this operation performs path pattern matching and solves context-free path reachability problem if necessary. This completes the part of the query execution plan building which concerns unnamed path patterns.

Another processing that occurs during the execution plan construction and was supported by us is related to named path patterns. They are processed independently of the unnamed path patterns found in MATCH clause and don't produce execution plan operations.

All named path patterns are collected from PATH PATTERN clauses. In the example query there is a path pattern *S = ()-/ [:c*

~S :d] | [:c (:y) :d] /->(). Then this named path patterns translated into algebraic expressions and stored in the corresponding global context of the query — *path pattern context*. This storage provides mapping between the path pattern name and its algebraic expression and its relation and source matrices. Initially these matrices are empty and populated later in the execution stage. For the example query this storage will be as follows:

$$\{S \rightarrow \{expr \rightarrow \mathcal{E}^c * Ref(S) * \mathcal{E}^d + \mathcal{E}^c * \mathcal{V}^y * \mathcal{E}^d,$$
$$m_{rel} \rightarrow \varnothing,$$
$$m_{src} \rightarrow \varnothing\}\}$$

Thus after execution plan building we receive *CFPQTraverse* operations that correspond to unnamed path patterns in MATCH clause and *path pattern context* that stores all named path patterns from PATH PATTERN clauses. Therefore we can proceed to the stage of execution plan evaluation.

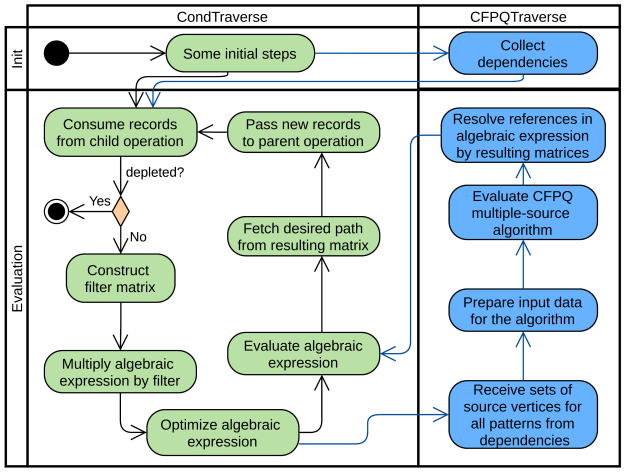

Figure 16: CFPQTraverse and CondTraverse evaluation

4.3.2 Execution plan evaluating. The remaining part of query processing is evaluation its execution plan. This section describes how the CFPQTraverse operation is performed. For explanation, we use example graph D_1 from Figure 1 and execution plan operations $LabelScan(n)$, $CondTraverse(r)$ and $CfpqTraverse(p)$ that were obtained in the previous section for example query from listing 7.

Let's first consider the structure of the execution plan operations. Operations have parent-child relationships, so they are formed into a tree. For example, the part of execution plan that derived from example query is shown in Figure 17. Each operation can consume a record from a child operation, process it and produce another one for the parent. Records contain information necessary for the parent operation, as well as everything to restore the response, such as identifiers of accumulated vertices and edges.

The CFPQTraverse operation is based on CondTraverse operation that already exists in the RedisGraph and performs a patterns matching. The activity diagram of this operations is shown in Figure 16 and described below. Actions that corresponds to CondTraverse operation are highlighted in green, actions of the CFPQTraverse operation that extend CondTraverse are highlighted in blue.

The CondTraverse works as follows. At first it consumes several records from the child operation and accumulates them in

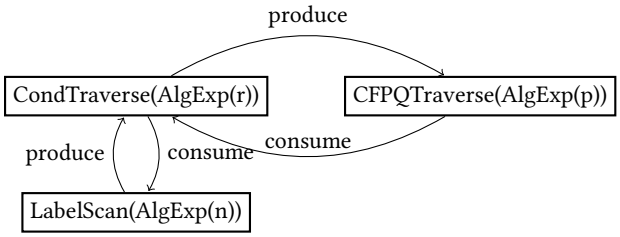

Figure 17: Example of part of the execution plan

the buffer. Here each record corresponds to the path that built by the child operation. For simplicity we can presume that each record is the destination vertex of the path. For example for the graph D_1 the *CondTraverse* from Figure 17 make *LabelScan* operation to produce vertices with the label X and then store the resulting set of vertices $\{1, 3\}$ in the buffer. The task of the CondTraverse is to continue the path from this vertices in such way that the resulting path satisfies pattern corresponding to this operation. To do this CondTraverse uses the algebraic expression obtained in the previous step. The resulting matrix of this expression represents all pairs of vertices between which there is a path satisfying the pattern. In order to find paths that start from given sources vertices CondTraverse uses a filter matrix. This matrix is constructed from the destination vertices retrieved from the record buffer and resembles matrix from boolean decomposition of label vertex matrix. For example filter matrix of set $\{1, 3\}$ is $r_f = \{(1, 1), (3, 3)\}$. Then algebraic expression $AlgExp(r)$ is multiplied to the left by r_f and we get algebraic expression $r_f * AlgExp(r) = r_f * (\mathcal{E}^a + \mathcal{E}^c)$. Then this expression is optimized, including all distributivity rules applying. For example, for expression $r_f * (\mathcal{E}^a + \mathcal{E}^r)$ after all optimizations we get expression $r_f * \mathcal{E}^a + r_f * \mathcal{E}^r$. Only after that this expression is evaluated and we get the matrix $\{(1, 2), (3, 5)\}$. This matrix exactly corresponds to all paths of length one where the source vertex is labelled by x and the edge is labeled by a or c. Then this paths are passed to the parent operation, in our case to $CFPQTraverse$, by producing new records.

The CFPQTraverse operation is arranged in the same way as CondTraverse but performs some additional work. Since each CFPQTraverse corresponds to path pattern, its algebraic expression may contain references to named path patterns. Therefore all named path patterns that the algebraic expression depends on must be processed. For this they are stored in the *set of operation dependencies* during its initialization. In this case, dependencies are extracted recursively, so that references inside named path patterns are also extracted. For example, path pattern p depends only on named path pattern S, so the dependency set of $CFPQTraverse(p)$ operation is $\{S\}$.

The CFPQTraverse execution stage starts the same way as CondTraverse one. First filter matrix is constructed from record buffer. For example, CFPQTraverse(p) consumes several records from CondTraverse(r), extracts from them the set of destination vertices $\{2, 5\}$ and builds filter matrix $p_f = \{(2, 2), (5, 5)\}$. Then this matrix is embedded in the algebraic expression of operation in the same way as in CondTraverse and we get new algebraic expression $p_f * AlgExp(p) = p_f * \mathcal{E}^b * Ref(S)$. After that for each reference in the algebraic expression we need to determinate the set of source vertices. This can be done during algebraic expression evaluation which we extended for this purpose.

Algorithm 8 Extension of multiplication evaluation

```
1: function EvalMul(e_l : AlgExp, e_r : AlgExp, context)
2:     M_l = Eval(e_l, context)
3:     if r is Ref(ref) then
4:         context[ref].src ← context[ref].src + getDst(M_l)
5:     ...                          ▷ Remaining original part of the EvalMul
```

Specifically, we extended the multiplication operation as shown in the listing 8. At first the left operand e_l of multiplication is evaluated by **Eval** function to matrix M_l in line 2. Then if the right operand of the multiplication is a reference, we need to populate source matrix of corresponding named path pattern. In this case matrix M_l specifies some set of destination vertices $\{j \mid M_l[i,j] = 1\}$ from which the path should continue. So we add this set to source matrix of named path pattern. For example, lets consider evaluation of $p_f * AlgExp(p) = p_f * \mathcal{E}^b * Ref(S)$. First, $p_f * \mathcal{E}^b$ is evaluated into matrix $M_l = \{(2,3),(2,6)\}$. It corresponds to edges starting from vertex 2 and labeled by b. Then destination vertices $\{3,6\}$ of M_l are extracted and added to source matrix of named path pattern S. So after this evaluation the path pattern context becomes the following:

$$\{S \rightarrow \{expr \rightarrow \mathcal{E}^c * Ref(S) * \mathcal{E}^d + \mathcal{E}^c * \mathcal{V}^y * \mathcal{E}^d,$$
$$m_{rel} \rightarrow \emptyset,$$
$$m_{src} \rightarrow \{(3,3),(6,6)\}\}\}$$

After that we have everything to run multiple-source CFPQ algorithm provided in listing 9 to resolve all operation dependencies. This algorithm is slightly different from *MultiSrcCFPQ* algorithm described in listing 2 and is a generalization of it. It receives the set of operation dependencies *deps* and path pattern context *context*. The algorithm's task is to populate relation and source matrices of all named path pattern from *deps*. To do this on each iteration for all pattern from *deps* this matrices are updated. Namely, first of all the algebraic expression is constructed from source matrix and algebraic expression of current pattern and then optimized in line **5**. This is followed by substitution references in line **6**, after which all references in the expression are replaced by relation matrices from *context*. At the end of iteration the resulting expression is evaluated and stored in relation matrix in line **7**. These iterations continue as long as the context changes, i.e. at least one of the pattern matrices m_{rel} or m_{src} changes during iteration.

Algorithm 9 Multiple-source context-free path querying algorithm in terms of algebraic expressions

```
1: function MultiSrcCFPQAlgExp(deps, context)
2:     while context is changing do
3:         for all p ∈ deps do
4:             src ← context[p].m_src
5:             expr ← Optimize(src * context[p].expr)
6:             FetchReferences(expr, ctx)
7:             context[p].m_rel ← Eval(expr)
```

After running this algorithm on the dependency set of *CFPQ-Traverse* operation and path pattern context we receive relation matrices of all named path pattern, on which operation depends. For example, after running it on dependency set $\{S\}$, relation and source matrices of pattern S will be as follows:

$$m_{rel} = \{(3,5),(3,6),(4,5),(4,6),(5,5),(5,6)\}$$
$$m_{src} = \{(3,3),(4,4),(5,5)\}$$

After that all references in the algebraic expression $p_f * AlgExp(p)$ are replaced with the relation matrices and we get algebraic expression $p_f * \mathcal{E}^b * context[S].m_{rel}$. Then this expression is evaluated to matrix $\{(3,5),(3,6)\}$ that corresponds to paths from 3rd vertex that satisfy the constraints specified by pattern S. Finally as well as CondTraverse, CFPQTraverse extracts desired paths from this resulting matrix and passes them to parent operation.

Therefore if we put together the results of all operations of execution plan the query from listing 7 on graph D_1 return the set of vertices $\{(1,5),(1,6)\}$.

4.4 Evaluation

In order to demonstrate applicability of the provided extension for RedisGraph we evaluate the proposed solution on the subset of cases provided in the section 3.2.

For RedisGraph evaluation, we used a PC with Ubuntu 18.04 installed. It has Intel Core i7-6700 CPU, 3.4GHz, and DDR4 64Gb RAM. RedisGraph with our extensions is installed form our GitHub repository[10].

4.4.1 Data preparing. We use the same graphs which are presented in table 1 to evaluate RedisGraph-based solution.

Graphs are loaded into RedisGraph database such that each vertex has a field `id` which value is unique and is in $[0 \ldots |V|-1]$, where $|V|$ is a number of vertices in the graph to load. This allows us to generate queries for specific chunk size using templates. The template for the g_1 query is provided in listing 10. Here `{id_from}` and `{id_to}` are placeholders for lower and upper bounds for `id`. The example of the exact query for chunk of size 16 is presented in listing 11.

Listing 10 Cypher query pattern for g_1

```
1: PATH PATTERN S =
       ()-/ [<:SubClassOf [~S | ()] :SubClassOf]
       | [<:Type [~S | ()] :Type] /->()
2: MATCH (src)-/ ~S /->()
3: WHERE {id_from} <= src.id and src.id <= {id_to}
4: RETURN count(*)
```

Listing 11 Query g_1 in Cypher using the template from listing 10

```
1: PATH PATTERN S =
       ()-/ [<:SubClassOf [~S | ()] :SubClassOf]
       | [<:Type [~S | ()] :Type] /->()
2: MATCH (src)-/ ~S /->()
3: WHERE 15 <= src.id and src.id <= 31
4: RETURN count(*)
```

Queries generator for all three queries (g_1, g_2, and *geo*) was implemented and used to create queries for all chunks which are used in the previous experiment.

[10]Sources of RedisGraph database with full-stack CFPQ support:https://github.com/YaccConstructor/RedisGraph/tree/path_patterns_dev. Access data: 19.07.2020.

4.4.2 Evaluation results. For evaluation we select *geo* query for *geospecies* graph as one of the hardest queries, and g_1 query for other graphs. Time and memory consumption are measured for each chunk processing. Results of time measurement are presented in figures 18–24.

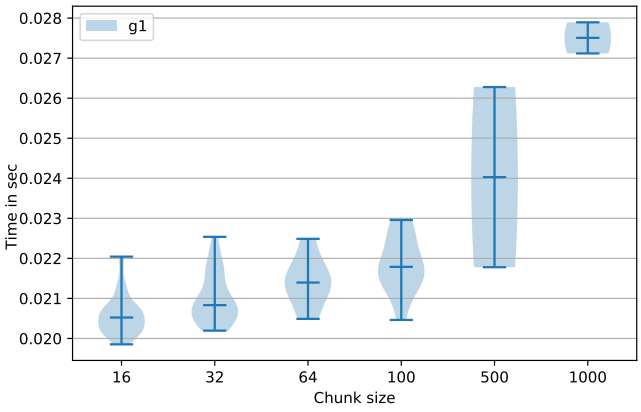

Figure 18: RedisGraph performance of *core* graph

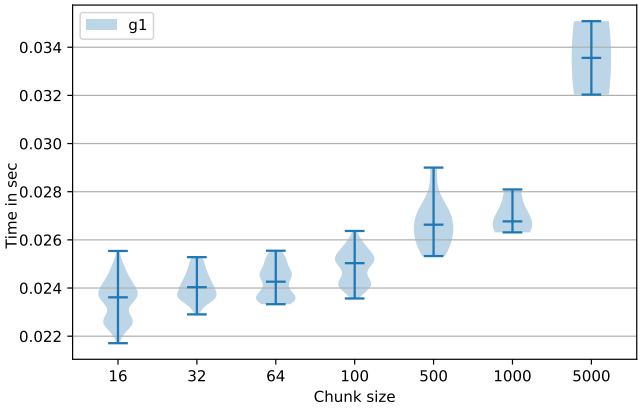

Figure 19: RedisGraph performance of *pathways* graph

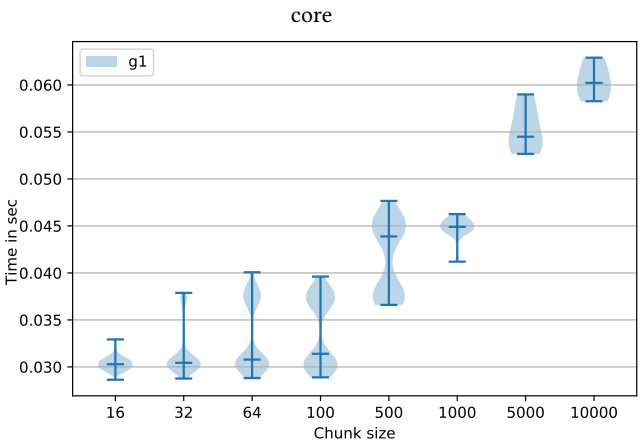

Figure 20: RedisGraph performance of *enzyme* graph

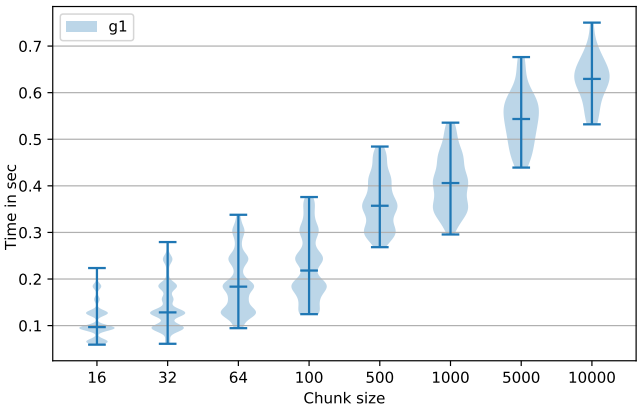

Figure 21: RedisGraph performance of *go* graph

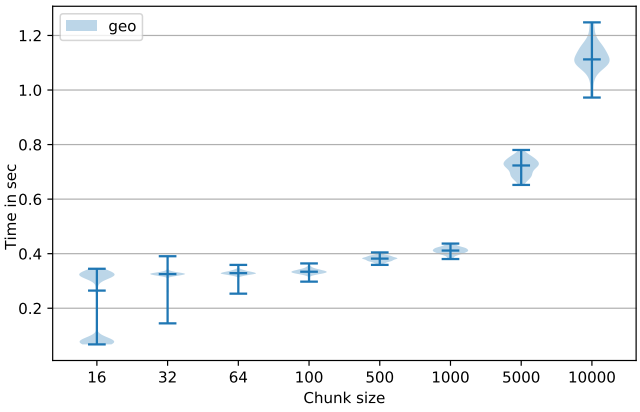

Figure 22: RedisGraph performance of *geospecies* graph

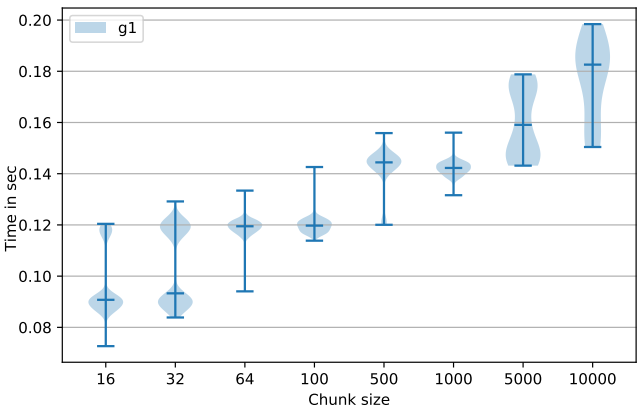

Figure 23: RedisGraph performance of *eclass_514en* graph

We can see, that results is comparable with one given in section 3.2. Processing time for all chunks, except chunk of size 10 000 for *geospecies* graph (fig. 22) is less then 1 second. Moreover, for chunks of size 16 processing median time is less then 0.1 second, except *geospecies* graph.

Memory consumption is presented in !!! La-la-la!!!!

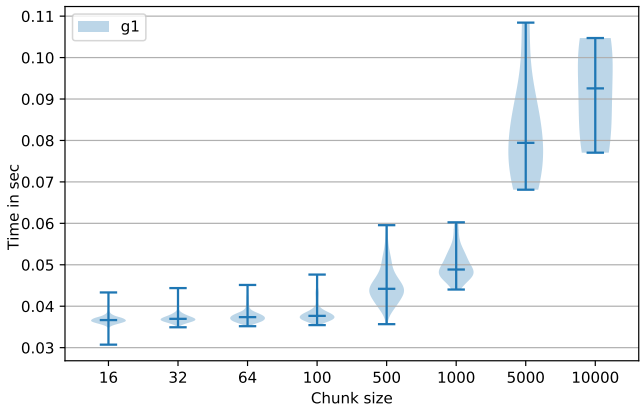

Figure 24: RedisGraph performance of *gohierarchy* graph

Table 2: Full graph processing time by RadisGraph with chunks of size !!!, time is measured in seconds (Chunks — the proposed solution, Full — results from [16])

Graph	#V	#E	Query	Chunks	Full
core	1323	3636	g_1	0.027	0.004
pathways	6238	18 598	g_1	0.028	0.011
gohierarchy	45 007	980 218	g_1	0.205	0.091
enzyme	48 815	117 851	g_1	0.058	0.018
eclass_514en	239 111	523 727	g_1	0.198	0.067
geospecies	450 609	2 311 461	geo	27.824	7.146
go	582 929	1 758 432	g_1	0.711	0.604

Additionally, we measure the time required to process full graph (to solve all-pairs reachability problem) by chunks of size !!!. Also, we compare our solution with results of Arseniy Terekhov et al. from [16] which were measured for RedisGraph deployed on the similar hardware and for the same graphs and queries. In [16] Azimov's algorithm was naively integrated with RedisGraph storage without support of query language and other mechanisms such as lazy query evaluation. Results are provide in the table 2.

We can see, that chunk-by-chunk processing is 2–7 times slower, but it is still require reasonable time. For example, it requires more than 200 times less time than solution of Jochem Kuijpers et al. [9] which is based on Neo4j and requires more than 6000 seconds. Moreover, while solution from [16] requires huge amount of memory (more than 16Gb for *geospecies* graph and *geo* query), our solution requires only !!! in the same scenario. Thus it is more suitable for general-purpose graph databases.

Finally we can conclude that provided

5 CONCLUSION

In this paper we propose a number of multiple-source modifications of Azimov's CFPQ algorithm. Evaluation of the proposed modifications on the real-world examples shows that queries results caching is not useful in evaluated scenarios and the naïve implementation is a best choice for integration with rel-world graph database. Finally, we provide the full-stack support of CFPQ. For our solution we implement corresponding Cypher extension as a part of `libcypher-parser`, integrate the proposed algorithm into RedisGraph, and extend RedisGraph execution plan builder to support extended Cypher queries. We demonstrate, that our solution is applicable for real-world graph analysis.

In the future, it is necessary to provide formal translation of Cypher to linear algebra, or find a maximal subset of Cypher which can be translated to linear algebra. There is a number of work on a subset of SPARQL to linear algebra translation, such as [6–8, 10]. But most of them practical-oriented and do not provide full theoretical basis to translate querying language to linear algebra. Other of them are discuss only partial cases and should be extended to cover real-world query languages. Deep investigation of this topic helps one to realize limits and restrictions of linear algebra utilization for graph databases. Moreover, it helps to improve existing solutions.

We show that evaluation of regular queries is possible in practice by using CFPQ algorithm, as far as regular queries is a partial case of the context-free one. But it seems, that the proposed solution is not optimal. For real-world solutions it is important to provide an optimal unified algorithm for both RPQ and CFPQ. One of possible way to solve this problem is to use tensor-based algorithm [13].

Another important task is to compare non-linear-algebra-based approaches to multiple-source CFPQ with the proposed solution. In [9] Jochem Kuijpers et al. show that all-pairs CFPQ algorithms implemented in Neo4j demonstrate unreasonable performance on real-world data. At the same time, Arseniy Terekhov et.al. shows that matrix-based all-pairs CFPQ algorithm implemented in appropriate linear algebra based graph database (RedisGraph) demonstrates good performance. But in the case of multiple-source scenario, when a number of start vertices is relatively small, non-linear-algebra-based solutions can be better, because such solutions naturally handle small required subgraph. Thus detailed investigation and comparison of other approaches to evaluate multiple-source CFPQ is required in the future.

REFERENCES

[1] R. Angles. 2018. The Property Graph Database Model. In *AMW*.

[2] Rustam Azimov and Semyon Grigorev. 2018. Context-free Path Querying by Matrix Multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18)*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3210259.3210264

[3] C. Barrett, R. Jacob, and M. Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (2000), 809–837. https://doi.org/10.1137/S0097539798337716 arXiv:https://doi.org/10.1137/S0097539798337716

[4] P. Cailliau, T. Davis, V. Gadepally, J. Kepner, R. Lipman, J. Lovitz, and K. Ouaknine. 2019. RedisGraph GraphBLAS Enabled Graph Database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 285–286.

[5] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (Dec. 2019), 25 pages. https://doi.org/10.1145/3322125

[6] Roberto De Virgilio. 2012. A Linear Algebra Technique for (de)Centralized Processing of SPARQL Queries. In *Conceptual Modeling*, Paolo Atzeni, David Cheung, and Sudha Ram (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 463–476.

[7] Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. 2019. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 27, 15 pages. https://doi.org/10.1145/3302424.3303962

[8] Fuad Jamour, Ibrahim Abdelaziz, and Panos Kalnis. 2018. A Demonstration of MAGiQ: Matrix Algebra Approach for Solving RDF Graph Queries. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1978–1981. https://doi.org/10.14778/3229863.3236239

[9] Jochem Kuijpers, George Fletcher, Nikolay Yakovets, and Tobias Lindaaker. 2019. An Experimental Study of Context-Free Path Query Evaluation Methods. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management (SSDBM '19)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/3335783.3335791

[10] Saskia Metzler and Pauli Miettinen. 2015. On Defining SPARQL with Boolean Tensor Algebra. *CoRR* abs/1503.00301 (2015). arXiv:1503.00301 http://arxiv.org/abs/1503.00301

[11] H. Miao and A. Deshpande. 2019. Understanding Data Science Lifecycle Provenance via Graph Segmentation and Summarization. In *2019 IEEE 35th*

International Conference on Data Engineering (ICDE). 1710–1713.

[12] Nikita Mishin, Iaroslav Sokolov, Egor Spirin, Vladimir Kutuev, Egor Nemchinov, Sergey Gorbatyuk, and Semyon Grigorev. 2019. Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication. In *Proceedings of the 2Nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'19)*. ACM, New York, NY, USA, Article 12, 5 pages. https://doi.org/10.1145/3327964.3328503

[13] Egor Orachev, Ilya Epelbaum, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying by Kronecker Product. In *Advances in Databases and Information Systems*, Jérôme Darmont, Boris Novikov, and Robert Wrembel (Eds.). Springer International Publishing, Cham, 49–59.

[14] Jakob Rehof and Manuel Fähndrich. 2001. Type-Base Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. *SIGPLAN Not.* 36, 3 (Jan. 2001), 54–66. https://doi.org/10.1145/373243.360208

[15] Petteri Sevon and Lauri Eronen. 2008. Subgraph Queries by Context-free Grammars. *Journal of Integrative Bioinformatics* 5, 2 (2008), 157 – 172. https://doi.org/10.1515/jib-2008-100

[16] Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3398682.3399163

[17] Xiaowang Zhang, Zhiyong Feng, Xin Wang, Guozheng Rao, and Wenrui Wu. 2016. Context-Free Path Queries on RDF Graphs. In *The Semantic Web – ISWC 2016*, Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda Gil (Eds.). Springer International Publishing, Cham, 632–648.

[18] Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/1328438.1328464