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 search for paths in edge-labeled graphs where constraints are formulated in terms of a formal language. The language restricts the set of accepted paths: the sentence formed by the labels of a path should be in the language. Regular languages are the most popular class of constraints used as navigational queries in graph databases. In some cases, regular languages are not expressive enough and context-free languages are used instead. Context-free path querying (CFPQ), can be used for RDF analysis [27], biological data analysis [22], static code analysis [20, 28], and in other areas.

CFPQ have been studied a lot since the problem was first stated by Mihalis Yannakakis in 1990 [26]. Jelle Hellings investigates various aspects of CFPQ in [8–10]. A number of CFPQ algorithms were proposed: (G)LL and (G)LR based algorithms by Ciro M. Medeiros et al. [15], Fred C. Santos et al. [21], Semyon Grigorev et al. [7], and Ekaterina Verbitskaia et al. [24]; CYK-based algorithm by Xiaowang Zhang et al. [27]; combinators-based approach to CFPQ by Ekaterina Verbitskaia et al. [25]. Nevertheless, the application of context-free constraints for real-world data analysis still faces many problems. The first problem is bad performance of the proposed algorithms on real-world data, as shown by Jochem Kuijpers et al. [14]. The second problem is that no graph database provides full-stack support of CFPQ, since most effort was made in developping algorithms and researching their theoretical properties. This fact hinders research of problems which can be reduced to CFPQ, thus it hinders the development of new solutions for them. For example, graph segmentation in data provenance analysis was recently reduced to CFPQ [17], but evaluation of the proposed approach was complicated because no graph database supported CFPQ.

Rustam Azimov proposed a matrix-based algorithm for CFPQ in [2]. This algorithm provides a solution performant enough for real-world data analysis, as shown by Nikita Mishim et al. in [18] and Arseniy Terekhov et al. in [23]. This algorithm computes reachability or provides a single path which satisfies constraints for *every* vertex pair in the graph. Namely it solves *all-pairs* context-free path querying problem. In many real-world scenarios it is redundant to handle all possible pairs, instead one can provide one or a relatively small set of start vertices.

While all-pairs context-free path querying is a problem well studied, there is no, best to our knowledge, solutions for the 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.

We also provide full-stack support of CFPQ for the RedisGraph[1] [4] graph database. We implement a Cypher query language extension[2] that makes it possible to use context-free constraints, and extend the RedisGraph to support this extension. As far as we know, it is the first full-stack implementation of CFPQ.

To sum up, we make the following contributions 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 is still based on linear algebra, hence it is simple to implementate and allows one to use high-performance libraries and utilize modern parallel hardware for queries evaluation.
(2) We evaluate two versions of the proposed algorithm: with caching of results and without caching (naive). Caching is

[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.

aimed to reduce repeated calculation of the same data. Our evaluation shows that the naive version is more performant and memory-efficient than the version with results caching in almost the all cases. We believe, it is a 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 extended Cypher with syntax for context-free constraints, implemented the proposed algorithm in a RedisGraph backend, and supported the 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 are used in this paper. Also, we provide a brief description of Azimov's algorithm which is used as a base of our solution.

2.1 Basic definitions of Graph Theory

In this paper we use a 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

- V is a finite 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.
- Σ_V and Σ_E are sets of labels of vertices and edges respectively, such that $\Sigma_V \cap \Sigma_E = \varnothing$.
- $\lambda_V : V \longrightarrow 2^{\Sigma_V}$ is a function that maps a vertex to a set of its labels, which can be empty.
- $\lambda_E : E \longrightarrow 2^{\Sigma_E} \setminus \{\varnothing\}$ is a function that maps an edge to a non-empty set of its labels, so each edge must have at least one label.

□

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

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.

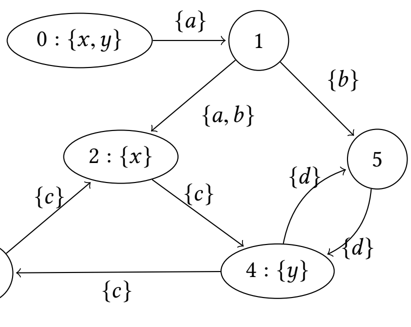

Figure 1: The input graph D_1

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 $\forall i, 0 \le i \le n : v_i \in V; \forall j, 1 \le j \le n : e_j = (v_j, v_{j+1}) \in E$.

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

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

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

□

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

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

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 matrix $\mathcal{E}^l$ of size $|V| \times |V|$, such that

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

□

Definition 2.5. A *boolean decomposition of adjacency matrix* M of the graph D is a set of Boolean matrices $\mathcal{E} = \{\mathcal{E}^l \mid l \in \Sigma_E\}$, where $\mathcal{E}^l$ is the adjacency matrix of label l. □

For example, the boolean decomposition of the adjacency matrix M of the graph D_1 is the set of Boolean matrices $\mathcal{E}^a$, $\mathcal{E}^b$, $\mathcal{E}^c$ and $\mathcal{E}^d$:

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

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

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

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

□

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 matrix $\mathcal{V}^l$ of size $|V| \times |V|$, such that

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

Definition 2.8. A *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 in this subsection we 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. A *context-free grammar* is a tuple $G = (N, \Sigma, P, S)$, where

- N is a finite set of nonterminals
- Σ is a finite set of terminals, $N \cap \Sigma = \varnothing$
- P is a finite set of productions of the form $A \to \alpha$, where $A \in N, \ \alpha \in (N \cup \Sigma)^*$
- S is the start nonterminal

□

Definition 2.10. A *context-free language* is a language generated by a context-free grammar G:

$$L(G) = \{w \in \Sigma^* \mid S \xRightarrow[G]{*} w\}$$

Where $S \xRightarrow[G]{*} w$ 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. A context-free grammar $G = (N, \Sigma, P, S)$ is in *Chomsky normal form* if every production in P has one of the following forms:

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

□

Definition 2.12. A context-free grammar $G = (N, \Sigma, P, S)$ is in *weak Chomsky normal form* if every production in P has one of the following forms:

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

□

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 occur in the right-hand side of productions.

The matrix-based CFPQ algorithms process grammars only in weak Chomsky normal form, but every context-free grammar can be transformed into the equivalent grammar in this form.

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

$$\begin{aligned} S &\to c \ S \ d \\ S &\to c \ y \ d \end{aligned} \tag{1}$$

This grammar generates the context-free language:

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

The following grammar G_1^{wcnf} is a result of the transformation of G_1 to weak Chomsky normal form:

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

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 relation $R_{G,D} \subseteq V \times V$:

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

where $l(\pi) \subset (\Sigma_V \cup \Sigma_E)^*$ is the set of 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, π is a path from vertex 3 to vertex 6 in the labeled graph presented in figure 1:

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

Labels along π form the set of sequences $l(\pi) = \{cy^n d \mid n \geq 0\}$. Only one of these sequences satisfies context-free constraints of the grammar G_1: cyd. The derivation of the sequence is the following:

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

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

Take a closer look at the definition of path labels, namely that it allows for zero or more repetitions of a label of each vertex. This makes it possible to omit vertex labels or, if there are many vertex labels, to use them in an arbitrary order. It also permits to write a query which uses one vertex label multiple times. This definition may appear strange in some cases, but it depends on the semantics of the graph query language. To formalize the semantics is future work, so we will stick to this definition in this paper.

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

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 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.

The context-free relation R_{G_1,D_1} for the graph D_1 and the context-free free grammar G_1 is the following:

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

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:

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

Definition 2.15. Suppose *Src* is a given set of start vertices, then *multiple-source context-free path querying problem* for the given *Src*, directed labeled graph D and context-free grammar G is to find a context-free relation

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

Thus we restrict start vertices of the paths of interest to be a vertices from the given set □

As a special case, a *single-source CFPQ* is when *Src* is a singleton set. If we set $Src = \{2\}$ in the previous example, then the result is:

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

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 presented in listing 1. Using Boolean matrices simplifies the implementation of the algorithm.

Algorithm 1 Context-free path querying algorithm

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

Note, that the provided algorithm computes not only the 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 form words derivable from any nonterminal in the given grammar. Also, this algorithm handles only the edge labels.

As was shown by Nikita Mishin et al. [18] and Arseniy Terekhov et al. [23], 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 path between them which forms a word in the given language. Thus it is impractical in cases when we are only interested in paths which start from the specific set of vertices, especially if this set is relatively small. Moreover, Azimov's algorithm operates over an adjacency matrice of the whole input graph, and as a result it requires a huge amount of memory, which may be a problem for a 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.

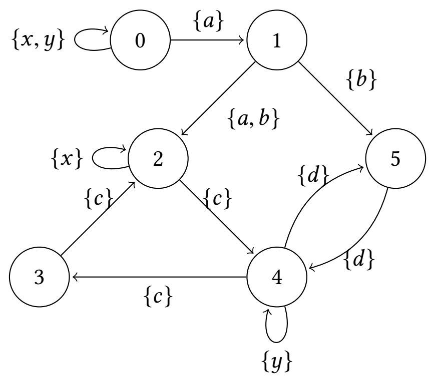

Figure 2: The example of D_1': the modified input graph D_1

The adjacency matrix M of the graph D_1' is

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

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(
   $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$,
   $G = (N, \Sigma, P, S)$,
   Src)
2: $T \leftarrow \{T^A \mid A \in N, T^A_{i,j} \leftarrow$ *false*, for all $i, j\}$
3: $TSrc \leftarrow \{TSrc^A \mid A \in N, TSrc^A_{i,j} \leftarrow$ *false*, for all $i, j\}$
4: **for all** $v \in Src$ **do** ▷ Input matrix initialization
5: $TSrc^S_{v,v} \leftarrow true$
6: **for all** $A \rightarrow x \in P$ **do** ▷ Simple rules initialization
7: **for all** $(v, to) \in E, \lambda_E(v, to) = x$ **do**
8: $T^A_{v,to} \leftarrow true$
9: **while** *T or TSrc is changing* **do** ▷ Algorithm's body
10: **for all** $A \rightarrow BC \in P$ **do**
11: $M \leftarrow TSrc^A * T^B$
12: $T^A \leftarrow T^A + M * T^C$
13: $TSrc^B \leftarrow TSrc^B + TSrc^A$
14: $TSrc^C \leftarrow TSrc^C +$ getDst(M)
15: **return** T^S
16: **function** getDst(M)
17: $A_{i,j} \leftarrow$ *false*
18: **for all** $(v, to) \in V^2 \mid M_{v,to} = true$ **do**
19: $A_{to,to} \leftarrow true$
20: **return** A

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

1: **function** CreateIndex(
   $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$,
   $G = (N, \Sigma, P, S)$)
2: $T \leftarrow \{T^A \mid A \in N, T^A_{i,j} \leftarrow$ *false*, for all $i, j\}$
3: $TSrc \leftarrow \{TSrc^A \mid A \in N, TSrc^A_{i,j}$*false*, for all $i, j\}$
4: **for all** $A \rightarrow x \in P$ **do** ▷ Simple rules initialization
5: **for all** $(v, to) \in E, \lambda_E(v, to) = x$ **do**
6: $T^A_{v,to} \leftarrow true$
7: **return** $(T, TSrc)$
8:
9: **function** MultiSrcCFPQSmart(
   $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$,
   $G = (N, \Sigma, P, S)$,
   Src,
   $Index = (T, TSrc)$)
10: $TNewSrc \leftarrow \{TNewSrc^A \mid A \in N, TNewSrc^A \leftarrow \varnothing\}$
11: **for all** $v \in Src \mid TSrc_{v,v} = false$ **do**
12: $TNewSrc^S_{v,v} \leftarrow true$
13: **while** *T or TNewSrc is changing* **do**
14: **for all** $A \rightarrow BC \in P$ **do**
15: $M \leftarrow TNewSrc^A * T^B$
16: $T^A \leftarrow T^A + M * T^C$
17: $TNewSrc^B \leftarrow TNewSrc^B + TNewSrc^A \setminus TSrc^B$
18: $TNewSrc^C \leftarrow TNewSrc^C +$ getDst(M) $\setminus TSrc^C$
return $TSrc^S * T^S$ ▷ We want to return only relevant data, not all cached results

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 $A \rightarrow BC \in P$ 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 start vertices for the current iteration (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.

In case when one have a sequence of similar queries to the single graph it may be useful to cache results of query evaluation and share them between queries. This may help to avoid recalculation of already calculated results. To introduce interqueries caching, we modify the previous version of algorithm. The modified 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 **11** of Algorithm 3. Thus, modified algorithm calculates paths from the particular vertex only once. Note, that CreateIndex function should be called first, after that the created index can be shared between multiple calls of MultiSrcCFPQSmart.

3.1 An Example

Consider first few steps of the proposed algorithm (without caching) on the graph D_1', grammar G_1^{wcnf}, and set of start vertices $Src = \{2\}$. At the first step (lines **4–5**) $TSrc^S_{2,2}$ sets to *true*, all other cells have value *false*. Then the set of matrices T is initialized using rules $C \rightarrow c, D \rightarrow d, Y \rightarrow y$ as follows:

$$T^C = \mathcal{E}^c = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}, T^D = \mathcal{E}^d = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \end{pmatrix},$$

$$T^Y = \mathcal{V}^y = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

At the first iteration of the while loop in line **9** for grammar rule $S \rightarrow CE$ the following computations take places. First, the matrix M is computed as follows

$$M = TSrc^S * T^C = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} * \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Since matrix T^E is empty, T^S is not updated in line **12**, and $TSrc^C$ is not updated in line **13**. But the matrix $TSrc^E$ is should be updated (line **14**):

$$TSrc^E = TSrc^E + \text{getDist}(M) =$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

This means that we are interested in paths that start from the vertex 4 and satisfy the constraints specified by nonterminal E.

The second rule is $S \longrightarrow CS_1$ and its processing is similar to previous one. After processing,

$$TSrc^{S_1} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

Third rule is $E \longrightarrow YD$. Here M is computed as follows:

$$M = TSrc^{E} * T^{Y} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} * \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

After that algorithm update T^E as follows:

$$T^{E} = T^{E} + M * T^{D} =$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} * \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Thus we know that there exists a path from vertex 4 to vertex 5 such that it forms a word derivable from E.

The last rule is $S_1 \longrightarrow SD$. During it processing since T^S is empty only $TSrc^S$ is updated:

$$TSrc^{S} = TSrc^{S} + TSrc^{S_1} =$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} * \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

This is the end of the first iteration.

At the second iteration, for rule $S \longrightarrow CE$ matrices M and then T^S are computed as follows:

$$M = TSrc^{S} * T^{C} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} * \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$T^{S} = M * T^{E} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} * \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Thus, we found the first path that satisfies our query. After all iterations finished, we get a final result:

$$T^{S} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

So, only vertices 4 and 5 are reachable from the vertex 2 by the path which forms word derivable form nonterminal S.

Table 1: Graphs for CFPQ evaluation

Graph	#V	#E	#subCalssOf	#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

3.2 Implementation Notes

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.

3.3 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.

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. 2) and g_2 (eq. 3) which are used in other works for CFPQ evaluation. Also we use *geo* (eq. 4) query which was provided by J. Kuijpers et. al [14] for *geospecies* RDF. Note that in queries we use $\overline{x}$ notation to denote inverse of x relation and respective edge.

$$\begin{aligned} S \longrightarrow & \overline{subClassOf}\ S\ subClasOf \mid \overline{type}\ \ S\ type \\ & \mid \overline{subClassOf}\ \ subClasOf \mid \overline{type}\ type \end{aligned} \tag{2}$$

$$S \longrightarrow \overline{subClassOf}\ S\ subClasOf \mid subClassOf \tag{3}$$

$$\begin{aligned} S \longrightarrow & broaderTransitive\ \ S\ \overline{broaderTransitive} \\ & \mid broaderTransitive\ \ \overline{broaderTransitive} \end{aligned} \tag{4}$$

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

[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

[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.

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 algorithm with caching we initialize cache once for each chunk size and accumulate results for all chunks for specific size.

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.

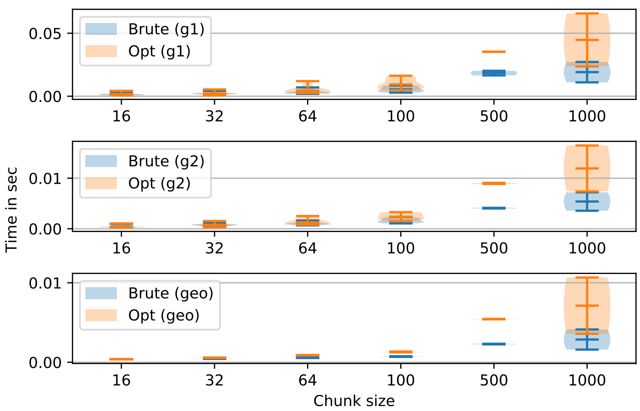

Figure 3: Performance of *core* graph querying

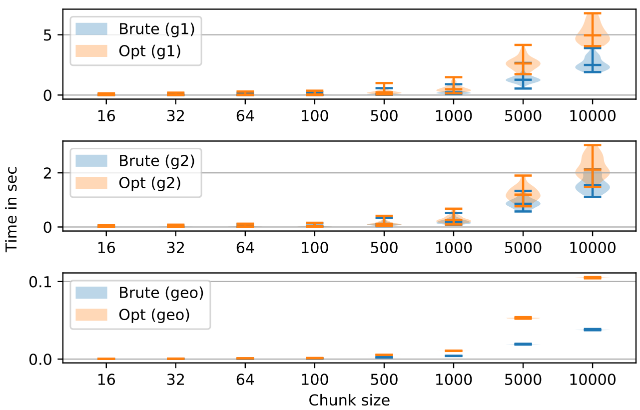

Figure 4: Performance of *go* graph querying

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 10 000 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

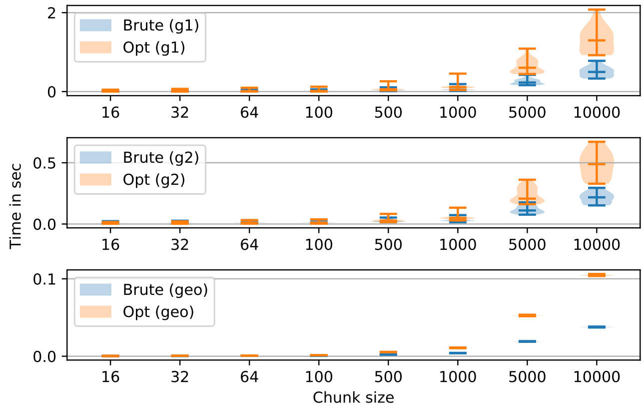

Figure 5: Performance of *eclass_514en* graph querying

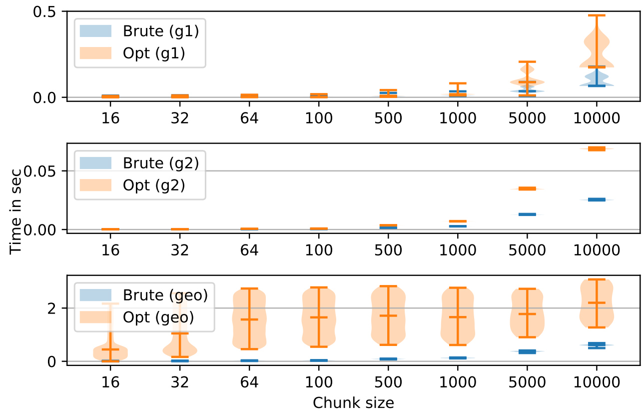

Figure 6: Performance of *geospecies* graph querying

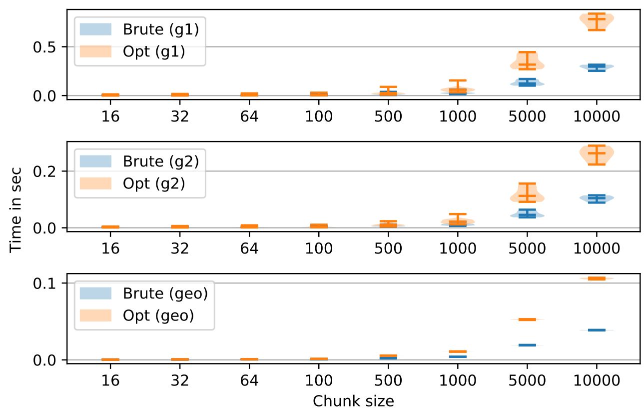

Figure 7: Performance of *enzyme* graph querying

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

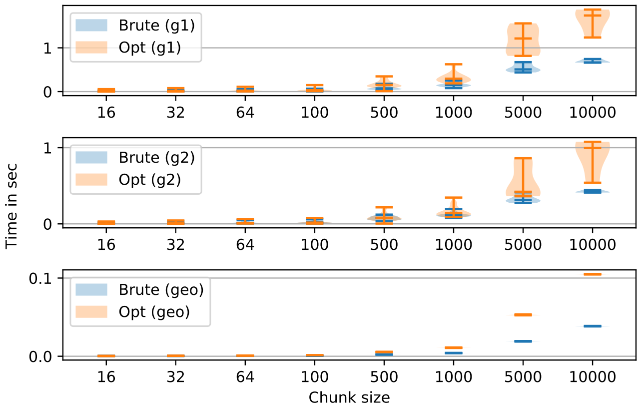

Figure 8: Performance of *gohierarchy* graph querying

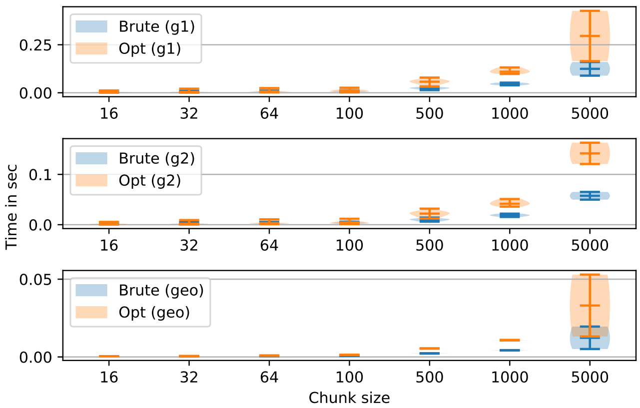

Figure 9: Performance of *pathways* graph querying

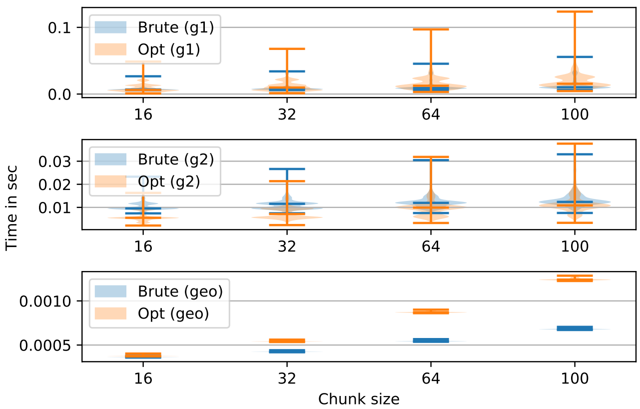

Figure 10: Performance of *eclass_514en* graph querying with small chunks

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 introduces significant overhead and does not lead to significant performance improvements. Also we can conclude that small

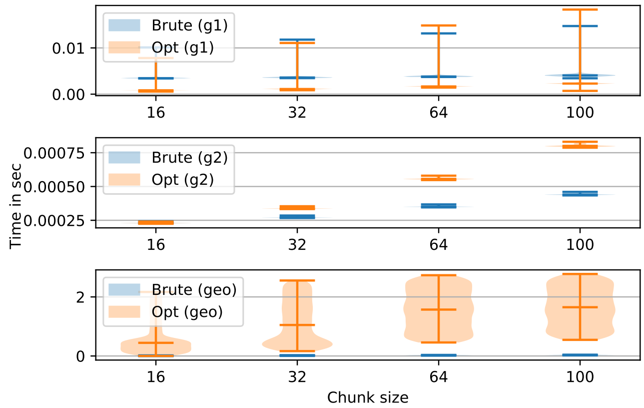

Figure 11: Performance of *geospecies* graph querying with small chunks

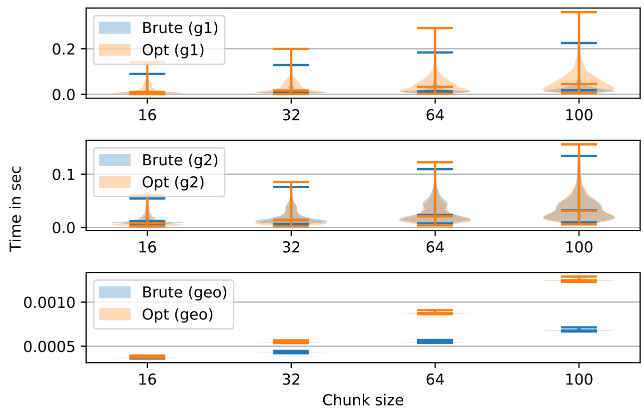

Figure 12: Performance of *go* graph querying with small chunks

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 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.

4 CFPQ FULL-STACK SUPPORT

In order to provide full-stack support of CFPQ it is necessatry to choose an appropriate graph database. It was shown by Arseniy Terekhov et al. in [23] 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. Like relationship patterns, they can be specified in the `MATCH` clause.

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. Named patterns can be defined in the `PATH PATTERN` clause. Using this feature, structure of query is pretty similar to context-free grammar in the Extended Backus-Naur Form (EBNF) [11].

Listing 4 Query based on example grammar G_1 (eq. 1) in Cypher with path patterns

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

The example of query which uses named path patters is presented in listing 4. This query is based on context-free grammar G_1 (eq. 1). Namely, path patter with name `S` specifies exactly the same constraint that specified by the grammar G_1. The `MATCH` clause uses pattern `S` in complex constraint which says that path of interest should starts in the vertex with label `x`, than in should goes throw edge with label `a` or `c`, and the end of path is a sequence of edges which starts from `b` and tail of this sequence matches with `S`.

For the example graph D_1 this query returns the next pairs of vertices `(v, to)` (as specified in `RETURN` clause): !!!!

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. Note that we implement[9] full extension, not only part which is necessary for simple CFPQ.

4.2 RedisGraph Extending

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 **??** 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.

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 **??**

[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.

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. 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.

The remaining part of query processing is evaluation its execution plan.

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 **??**. 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.

4.3 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 3.3.

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].

4.3.1 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 $[0 \dots |V|-1]$, 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 5. 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 6.

Listing 5 Cypher query pattern for g_1

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

Listing 6 Query g_1 in Cypher using the template from listing 5

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

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

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.

4.3.2 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 13–19.

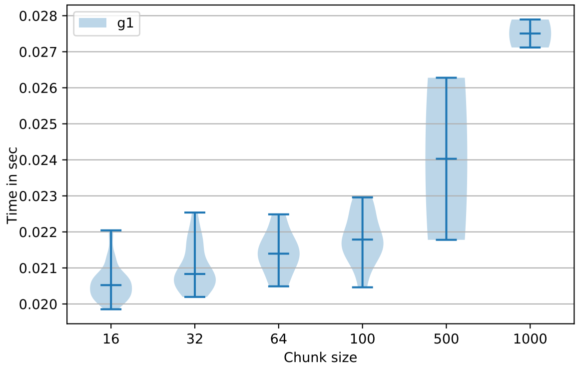

Figure 13: RedisGraph performance on *core* graph

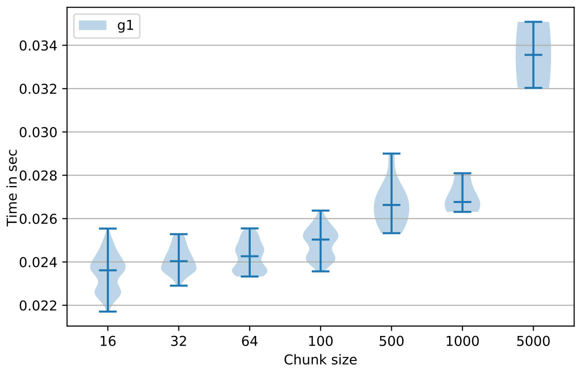

Figure 14: RedisGraph performance on *pathways* graph

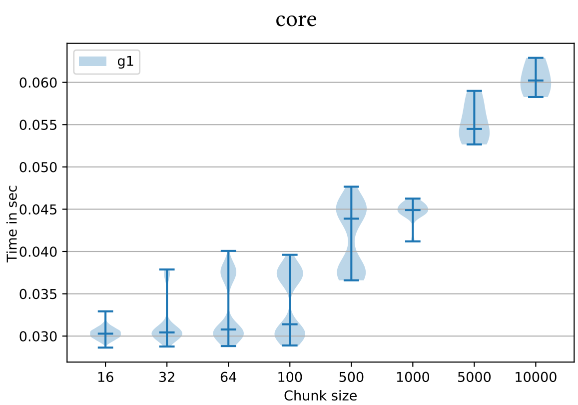

Figure 15: RedisGraph performance on *enzyme* graph

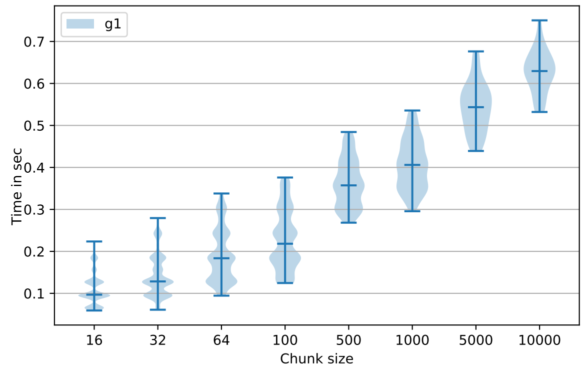

Figure 16: RedisGraph performance on *go* graph

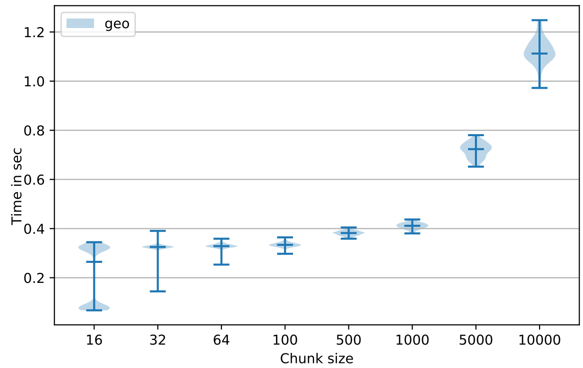

Figure 17: RedisGraph performance on *geospecies* graph

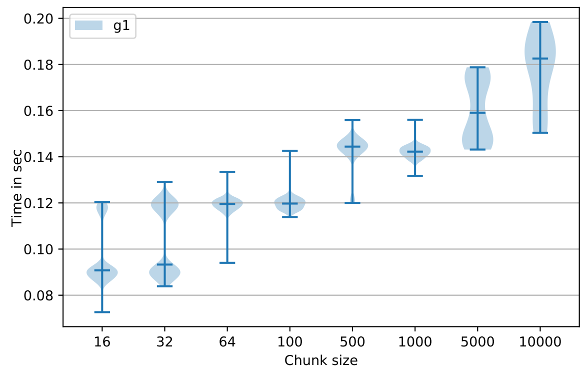

Figure 18: RedisGraph performance on *eclass_514en* graph

We can see, that results is comparable with one given in section 3.3. Processing time for all chunks, except chunk of size 10 000 for *geospecies* graph (fig. 17) 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 for two big graphs *eclass_514en* and *geospecies* is presented in figures 20 and 21 respectively. We can see, that amount of used memory depends on graph and query, but for relatively small chunks ($\leq$ 1000) RedisGraph uses less that 50Mb of RAM to process one chunk. Note that RedisGraph includes memory management system, thus in our experiments

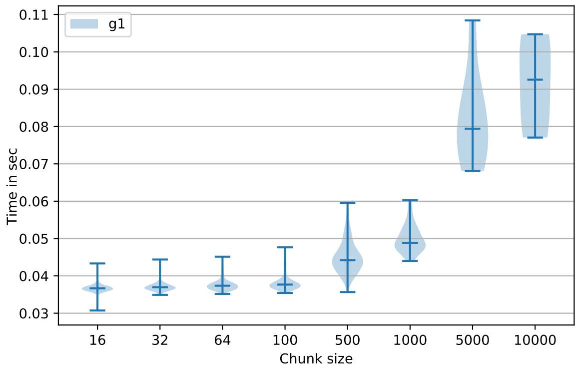

Figure 19: RedisGraph performance on *gohierarchy* graph

all allocated memory is measured, not only really used for query evaluation. As a result, we can conclude that multiple-source CFPQ is significantly more memory efficient than creation of full reachability index and its filtering: processing the chunk of size 10 000 on *geospecies* graph requires less than 200Mb, while full index creation requires 16Gb [23].

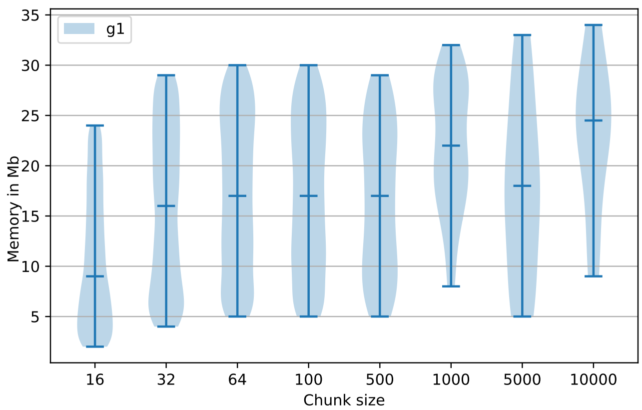

Figure 20: RedisGraph memory consumption on *eclass_514en* graph

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 [23] which were measured for RedisGraph deployed on the similar hardware and for the same graphs and queries. In [23] 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. [14] which is based on Neo4j and requires more than 6000 seconds. Moreover, while solution from [23] 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

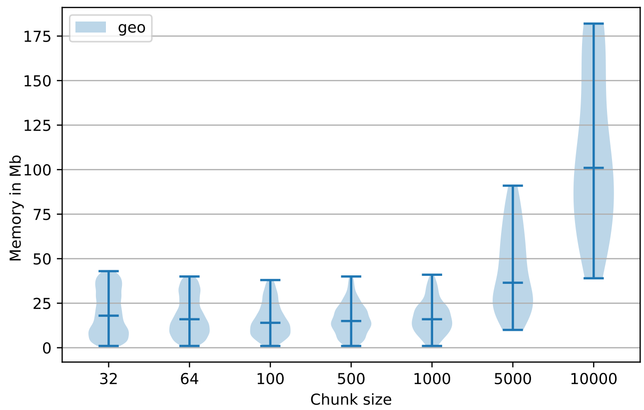

Figure 21: RedisGraph memory consumption on *geospecies* graph

Table 2: Full graph processing time by RadisGraph with chunks of size !!!, time is measured in seconds (Chunks — the proposed solution, Full — results from [23])

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

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, 12, 13, 16]. 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 [19].

Another important task is to compare non-linear-algebra-based approaches to multiple-source CFPQ with the proposed solution. In [14] 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.

ACKNOWLEDGEMENTS

Grants

Thanks to Ekaterina Verbitskaia for paper improvements and fruitful discussion.

Thanks to Roi Lipman for his help with RedisGraph internals investigation and for his comment on impractical memory consumption of the original Aziomv's algorithm which motivates us to develop the described solution.

Anonimus reviewers !!!

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] Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free Path Querying with Structural Representation of Result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3166094.3166104

[8] Jelle Hellings. 2014. Conjunctive context-free path queries. In *Proceedings of ICDT'14*. 119–130.

[9] Jelle Hellings. 2015. Path Results for Context-free Grammar Queries on Graphs. *CoRR* abs/1502.02242 (2015). arXiv:1502.02242 http://arxiv.org/abs/1502.02242

[10] Jelle Hellings. 2015. Querying for Paths in Graphs using Context-Free Path Queries. *arXiv preprint arXiv:1502.02242* (2015).

[11] ISO/IEC. 1996. International Standard EBNF Syntax Notation. http://www.iso.ch/cate/d26153.html. 14977 edn. Online; accessed 19.07.2020.

[12] 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

[13] 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

[14] 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

[15] Ciro M. Medeiros, Martin A. Musicante, and Umberto S. Costa. 2018. Efficient Evaluation of Context-free Path Queries for Graph Databases. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. ACM, New York, NY, USA, 1230–1237. https://doi.org/10.1145/3167132.3167265

[16] 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

[17] 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.

[18] 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

[19] 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.

[20] 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

[21] Fred C. Santos, Umberto S. Costa, and Martin A. Musicante. 2018. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 225–233.

[22] 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

[23] 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

[24] Ekaterina Verbitskaia, Semyon Grigorev, and Dmitry Avdyukhin. 2016. Relaxed Parsing of Regular Approximations of String-Embedded Languages. In *Perspectives of System Informatics*, Manuel Mazzara and Andrei Voronkov (Eds.). Springer International Publishing, Cham, 291–302.

[25] Ekaterina Verbitskaia, Ilya Kirillov, Ilya Nozkin, and Semyon Grigorev. 2018. Parser Combinators for Context-free Path Querying. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala (Scala 2018)*. ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/3241653.3241655

[26] Mihalis Yannakakis. 1990. Graph-theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*. ACM, New York, NY, USA, 230–242. https://doi.org/10.1145/298514.298576

[27] 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.

[28] 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