GLL-based Context-Free Path Querying for Neo4j

Vadim Abzalov,[1, *] Vlada Pogozhelskaya,[2, **] Vladimir Kutuev,[1, *] Olga Bachishche,[3, ****] and Semyon Grigorev[1, *****]**

[1] *Saint Petersburg State University, St. Petersburg, Russia*
[2] *National Research University Higher School of Economics, St. Petersburg, Russia*
[3] *ITMO University, St. Petersburg, Russia*

Abstract—We propose a GLL-based *context-free path querying* algorithm that handles queries in Extended Backus-Naur Form (EBNF) using Recursive State Machines (RSM). Utilization of EBNF allows one to combine traditional regular expressions and mutually recursive patterns in constraints natively. The proposed algorithm solves both the *reachability-only* and the *all-paths* problems for the *all-pairs* and the *multiple sources* cases. The evaluation on real-world graphs demonstrates that the utilization of RSMs increases the performance of query evaluation. Being implemented as a stored procedure for Neo4j, our solution demonstrates better performance than a similar solution for RedisGraph. The performance of our solution on regular path queries is comparable to the performance of the native Neo4j solution, and in some cases, our solution requires significantly less memory.

Keywords and phrases: *Graph database, graph querying, context-free path querying, CFPQ, reachability problem, all-paths problem, generalized LL, GLL*

1. INTRODUCTION

Context-free path querying (CFPQ) allows one to use context-free grammars to specify constraints on paths in edge-labeled graphs. Compared to regular path queries (RPQ), CFPQ is strictly more expressive: for instance, a context-free grammar can be built to find siblings or same-generation categories in a taxonomy [1, 2]. This expressiveness allows CFPQ to be used for graph analysis in such areas as bioinformatics (hierarchy analysis [3], similarity queries [4]), data provenance analysis [5], static code analysis [6, 7]. Although a lot of algorithms for CFPQ have been proposed, poor performance on real-world graphs and bad integration with real-world graph databases and graph analysis systems are still problems that hinder the adoption of CFPQ.

The problem with the performance of CFPQ algorithms in real-world scenarios was pointed out by Jochem Kuijpers [2] as a result of an attempt to extend the Neo4j graph database with CFPQ. Several algorithms, based on such techniques as LR parsing algorithm [8], dynamic programming [9], LL parsing algorithm [10], linear algebra [1], were implemented using Neo4j as a graph storage and evaluated. None of them were performant enough to be used in real-world applications.

Since Jochem Kuijpers pointed out the performance problem, it was shown that linear algebra based CFPQ algorithms, which operate over the adjacency matrix of the input graph and utilize parallel algorithms for linear algebraic operations, demonstrate good performance [11]. Moreover, the matrix-based CFPQ algorithm is a base for the first full-stack support of CFPQ by extending the RedisGraph graph database [11].

However, adjacency matrix is not the only possible format for graph representation, and data format conversion may take a lot of time; thus, it is not applicable in some cases. As a result, the development of a performant CFPQ algorithm for graph representations not based on matrices and its integration with real-world graph databases is still an open problem. Moreover, while the *all pairs context-free constrained reachability* is widely discussed in the community, such practical cases as the *all-paths* queries and the *multiple sources* queries are not studied enough.

[*] E-mail: vadim.i.abzalov@gmail.com
[**] E-mail: vvpogozhelskaia@edu.hse.ru
[***] E-mail: v.kutuev@spbu.ru
[****] E-mail: bachisheo@gmail.com
[*****] E-mail: s.v.grigoriev@spbu.ru

Additionally, almost all existing solutions are for *reachability-only* problem. Recently, Jelle Helligs in [12] and Rustam Azimov in [13] proposed algorithms that allow one to extract paths that satisfy the specified context-free constraints. The ability to extract paths of interest is important for some applications where the user wants to know not only the fact that one vertex is reachable from another, but also to get a detailed explanation of why this vertex is reachable. One of such applications is program code analysis where the fact is a potential problem in code, and paths can help to analyze and fix this problem. While the utilization of general-purpose graph databases for code analysis is gaining popularity [14], CFPQ algorithms that provide a structural representation of paths are not studied enough.

Generalized LL (GLL) is a parsing algorithm that can handle any context-free grammar, including left-recursive and ambiguous ones. Similar to GLR (e.g. Tomita algorithm), it achieves this by using a graph-structured stack (GSS) to efficiently share and manage the parsing state, allowing it to explore all possible derivations in parallel without an exponential time cost. The output of the parsing is a compact representation of all possible parse trees for the input, known as a "parse forest". It was shown that the GLL algorithm can be naturally generalized to the CFPQ algorithm [15]. Moreover, this provides a natural solution not only for the *reachability-only* problem but also for the *all-paths* problem. At the same time, there exists a high-performance GLL parsing algorithm [16] and its implementation in the Iguana project [17].

Additionally, pure context-free grammars in Backus-Naur Form (BNF) are too verbose to express complex constraints. However, almost all algorithms require a query to be in such form. At the same time, Extended Backus-Naur Form (EBNF) can be used to specify context-free languages. EBNF allows combining typical regular expressions with mutually recursive rules which are required to specify context-free languages. That makes EBNF more user-friendly. But there are no CFPQ algorithms that utilize EBNF for query specification.

In this paper, we generalize the GLL parsing algorithm to handle queries in EBNF without its transformation. We show that it allows us to increase the performance of query evaluation. We also integrate our solution with the Neo4j graph database and evaluate it. Thus, we make the following contributions in this paper.

- We propose the GLL-based CFPQ algorithm that can handle queries in Extended Backus-Naur Form without transformation. Our solution utilizes Recursive State Machines (RSM) [18] for it. The proposed algorithm can be used to solve both *reachability-only* and *all-paths* problems.

- We provide an implementation of the proposed CFPQ algorithm. By experimental study on real-world graphs we show that utilization of RSMs allows one to increase performance of query evaluation.

- We integrate the implemented algorithm with Neo4j by providing a respective stored procedure that can be called from the Cypher. Currently, we use Neo4j as a graph storage and do not extend Cypher to express context-free path patterns. So, expressive power of our solution is limited: we cannot use the full power of Cypher within our constraints. Implementing a query language extension amounts to a lot of additional effort and is a part of future work.

- We evaluate the proposed solution on several real-world graphs. Our evaluation shows that the proposed solution in order of magnitude faster than a similar linear algebra-based solution for RedisGraph. Moreover, we show that our solution is compatible with native Neo4j solution for RPQs, and in some cases requires significantly less memory. Note that while Cypher expressively is limited, our solution can handle arbitrary RPQs.

The paper has the following structure. In section 2 we introduce basic notions and definitions from graph theory and formal language theory. Then, in section 3 we introduce *recursive state machines* and provide an example of CFPQ evaluation using naïve strategy which may leads to infinite computations. Section 4 contains a description of the GLL-based CFPQ evaluation algorithm which uses RSM and solves problems of naïve strategy from the previous section: the algorithm always terminates and can build a finite representation of all paths of interest. Section 5 introduces a data set (both graphs and queries) which will be used further for experiments. Further, in section 6 we use this data set to compare different versions of the GLL-based CFPQ algorithm. After that, in section 7 we provide details on the integration of the best version into the Neo4j graph database, and evaluate our solution on the data set introduced before. Related work is discussed in section 8. Final remarks and conclusion are provided in section 9.

2. PRELIMINARIES

In this section, we introduce basic definitions and notations for graphs, regular languages, context-free grammars, and finally, we formulate variants of the FLPQ problem.

We use a directed edge-labelled graph as a data model. We denote a graph as $D = \langle V, E, L\rangle$, where V is a finite set of vertices, $E \subseteq V \times L \times V$ is a set of edges, and L is a set of edge labels. A path π in a graph D is a sequence of edges: $u \xrightarrow{l_0} \ldots \xrightarrow{l_m} v$. We denote a path between vertices u and v as $u\pi v$. The function ω maps a path to a word by concatenating the labels of its edges:

$$\omega(\pi) = \omega(u \xrightarrow{l_0} \ldots \xrightarrow{l_m} v) = l_0 \ldots l_m. \tag{1}$$

In the context of formal languages, we would like to remind the reader of some fundamental facts about regular expressions and regular languages.

Definition 1. *A regular expression over the alphabet Σ specifies a regular language using the following syntax.*

- *If $t \in \Sigma$, then t is a regular expression.*
- *The concatenation $E_1 \cdot E_2$ of two regular expressions E_1 and E_2 is a regular expression. Note that in some cases $\cdot$ may be omitted.*
- *The union $E_1 \mid E_2$ of two regular expressions E_1 and E_2 is a regular expression.*
- *The Kleene star $E^* = \bigcup_{n=0}^{n=\infty} R^n$ of a regular expression E is a regular expression.*

In some cases E^+ can be used as syntactic sugar for $E \cdot E^*$.

Both regular expressions and finite automata specify regular languages and can be converted into each other. Every regular language can be specified using deterministic finite automata without ε-transitions. Note that regular languages form a strong subset of context-free languages.

Definition 2. *A context-free grammar is a 4-tuple $G = \langle \Sigma, \mathcal{N}, P, S\rangle$, where Σ is a finite set of terminals, $\mathcal{N}$ is a finite set of nonterminals, $S \in N$ is the start nonterminal, and P is a set of productions. Each production has the following form: $N_i \to w$, where $N_i \in \mathcal{N}$ is the left-hand side of the production, and $w \in (\Sigma \cup \mathcal{N})^*$ is the right-hand side of the production.*

For simplicity, $S \to w_1 \mid w_2$ is used instead of $S \to w_1; S \to w_2$. Some example grammars are presented in equations 22, 21, and 23.

Definition 3. *A derivation step (in the grammar $G = \langle \Sigma, \mathcal{N}, P, S\rangle$) is a production application: having a sequence of form $w_1 N_0 w_2$, where $N_0 \in \mathcal{N}$ and $w_1, w_2 \in (\Sigma \cup \mathcal{N})^*$, and a production $N_0 \to w_3$, one gets a sequence $w_1 w_3 w_2$, by replacing the left-hand side of the production with its right-hand side.*

Definition 4. *A word $w \in \Sigma^*$ is derivable in the grammar $G = \langle \Sigma, \mathcal{N}, P, S\rangle$ if there is a sequence of derivation steps such that the initial sequence is the start nonterminal of the grammar and w is a final sequence: $S \to w_1 \to \ldots \to w$, or $S \to^* w$.*

Definition 5. *The language specified by the context-free grammar $G = \langle \Sigma, \mathcal{N}, P, S\rangle$ (denoted $\mathcal{L}(G)$) is a set of words derivable from the start nonterminal of the given grammar: $\mathcal{L}(G) = \{w \mid S \to^* w\}$.*

Definition 6. *A context-free grammar $G = \langle \Sigma, \mathcal{N}, P, S\rangle$ is in Extended Backus-Naur Form (EBNF) if productions have the form $\mathcal{N} \to E$ where E is a regular expression over $\mathcal{N} \cup \Sigma$.*

Grammar in EBNF can be converted to Backus-Naur Form, but such transformation requires introducing new productions and new nonterminals, which leads to a significant increase in the size of the grammar and, subsequently, poor performance of path querying algorithms.

We use a generalization of a *parse tree* to represent the result of a query.

Definition 7. *Parse tree of the sequence w w.r.t. the context-free grammar $G = \langle \Sigma, \mathcal{N}, P, S \rangle$ in EBNF is a rooted, node-labelled, ordered tree with the following properties.*

- *All leaves are labelled with the elements of $\Sigma \cup \{\varepsilon\}$, where $\varepsilon \notin \Sigma \cup \mathcal{N}$ is a special symbol to demote the empty string. Left-to-right concatenation of leaves forms w.*
- *All other nodes are labelled with the elements of $\mathcal{N}$.*
- *The root is labelled with S.*
- *Let w_0 is an ordered concatenation of child's labels of node labelled with N_i. Then $w_0 \in \mathcal{L}(r)$, where $N_i \to r \in P$.*

A set of problems for the graph D and the language $\mathcal{L}$ can be formulated:

$$R = \{(u, v) \mid \text{exists a path } u\pi v \text{ in } D, \omega(\pi) \in \mathcal{L}\} \tag{2}$$

$$Q = \{\pi \mid \text{exists a path } u\pi v \text{ in } D, \omega(\pi) \in \mathcal{L}\} \tag{3}$$

$$R(I) = \{(u, v) \mid \text{exists a path } u\pi v \text{ in } D, u \in I, \omega(\pi) \in \mathcal{L}\} \tag{4}$$

$$Q(I) = \{\pi \mid \text{exists a path } u\pi v \text{ in } D, u \in I, \omega(\pi) \in \mathcal{L}\} \tag{5}$$

Here 2 is the *all-pairs reachability problem*, and 3 is the *all-pairs all-paths problem.* The additional parameter I denotes the set of start vertices in 4 and 5. Thus 4 and 5 — are the *multiple source reachability* and the *multiple source all-paths* problems, respectively. Note that for the *all-paths* problems, the result Q can be an infinite set. Typically, the algorithms for these problems build a finite structure which contains all paths of interest, not the explicit set of paths. As far as the intersection of a regular and a context-free language is context-free [19], the mentioned finite representation of the result for the reachability problem is a representation of a context-free language.

This work is focused on a context-free language, but the partial case — regular languages — is also discussed. This corresponds to two specific cases of formal language path querying (FLPQ): *Context-Free Path Querying* (CFPQ) and *Regular Path Querying* (RPQ).

3. RECURSIVE STATE MACHINES

Recursive state machine or RSM [18] is a way to represent context-free languages in a way resembling finite automata. It allows us to use a graph-based representation of context-free languages specification and unify the processing of regular and context-free languages. We will use the following definition of RSM in this work.

Definition 8. *Recursive state machine (RSM) is a tuple $\mathcal{R} = \langle \mathcal{N}, \Sigma, B, B_S, Q, Q_S \rangle$ where:*

- *N is a set of nonterminal symbols,*
- *Σ is a set of terminal symbols,*
- *Q is a set of states of the RSM,*
- *Q_S is a set of the start states of all boxes,*
- *$B = \{B_{N_i} \mid N_i \in \mathcal{N}\}$ is a set of boxes, where each $B_{N_i} = \langle Q_{N_i}, q_S, Q_F^{N_i}, \delta \rangle$ is a deterministic finite automaton without ε-transitions called a box. Here:*
  - *$Q_{N_i} \subseteq Q$ is a set of states of the box,*

  - $q_S \in Q_{N_i} \cap Q_S$ *is the start state of the box,*
  - $Q_F^{N_i} \subseteq Q_{N_i}$ *is a set of final states of the box,*
  - $\delta \subseteq Q_{N_i} \times (\Sigma \cup Q_S) \times Q_{N_i}$ *is a transition function of the box.*

- $B_S \in B$ *is the start box of the RSM.*

Thus, RSM is a set of deterministic finite state machines without ε-transitions over the alphabet $\Sigma \cup Q_S$. But the computation process of RSM differs from that of DFA because it involves using a stack to handle transitions labelled with elements of Q_S.

We can use RSM to compute the reachable vertices in a graph. To describe the current state of the computation, we use *configurations*. Computation can be defined as a transition between configurations.

Definition 9. *A Configuration $C_{\mathcal{R}}$ of the computation of the RSM $\mathcal{R} = \langle \mathcal{N}, \Sigma, B, B_S, Q, Q_S \rangle$ over the graph $D = \langle V, E, L \rangle$ is a tuple $(q, v, \mathcal{S})$ where:*

- $q \in Q$ *is a current state of RSM,*
- $\mathcal{S}$ *is the current stack, whose frames have one of two types:*
  - *return addresses frame (elements of Q) to specify states to continue computation after the call is finished;*
  - *parsing tree node to store fragments of a parsing tree,*
- $v \in V$ *is the current vertex (current position in the input).*

Definition 10. *A transition step of the RSM specifies how to get new configurations of RSM, given the current configuration. $C_{\mathcal{R}} \vdash W$ denotes that $\mathcal{R}$ can go to each configuration in W from the configuration $C_{\mathcal{R}}$.*

$$\begin{aligned}(q, v, w_0 :: s :: \mathcal{S}) \vdash & \{(q', v', Node(t) :: w_0 :: s :: \mathcal{S}) \mid (q, t, q') \in \delta, (v, t, v') \in E\} \\ & \cup \{(s', v, q' :: w_0 :: s :: \mathcal{S}) \mid (q, s', q') \in \delta, s' \in Q_S\} \\ & \cup \{(s, v, Node(N_i, w_0) :: \mathcal{S}) \mid q \in Q_F^{N_i}\}\end{aligned}$$

where w_0 is a possibly empty sequence of terminal and nonterminal nodes, $Node(t)$ and $Node(N_i, w_0)$ — terminal and nonterminal nodes respectively, q', s — return addresses (states).

To simplify the acceptance condition, we introduce the concept of an *extended RSM.*

Definition 11. *For the given RSM $\mathcal{R} = \langle \mathcal{N}, \Sigma, B, B_S, Q, Q_S \rangle$, the extended RSM*

$$\mathcal{R}' = \langle \mathcal{N} \cup \{S'\}, \Sigma \cup \{\$\}, B \cup B'_S, B'_S, Q \cup \{q'_0, q'_1, q'_2\}, Q_S \cup \{q'_0\} \rangle$$

is an RSM that defines the same language and is built from $\mathcal{R}$ by adding a new start box

$$B'_S = \langle \{q'_0, q'_1, q'_2\}, q'_0, \{q'_2\}, \{(q'_0, q_0, q'_1), (q'_1, \$, q'_2)\} \rangle$$

where:

- q_0 *is a start state of B_S,*
- *\$ is a special symbol to mark the end of input, $\$ \notin \Sigma$,*
- q'_i *are newly added states, $q'_i \notin Q$.*

Finally, for the given extended RSM $\mathcal{R}$ and the given graph D we say that v_n is reachable from v_0 w.r.t. $\mathcal{R}$ if $(q'_0, v_0, []) \vdash^* C$ such that $(q'_1, v_n, [N_S]) \in C$, where $\vdash^*$ denotes zero or more transition steps, and N_S is a node for the start nonterminal of the original (not extended) RSM. Additionally, N_S represents a respective path.

Now we need a way to compute transitions and to build trees efficiently, avoiding recomputation and infinite cycles that are possible with the naive implementation. Moreover, we need a compact representation of all paths of interest whose number can be infinite.

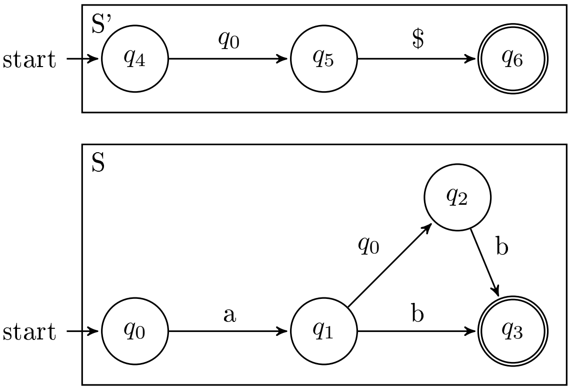


Figure 1. Extended RSM for grammar $S \to a\ b \mid a\ S\ b$.

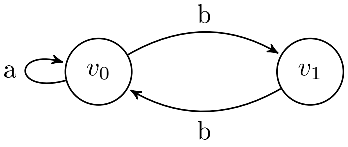


Figure 2. Input graph.

3.1. Example

In this section, we introduce a step-by-step example of the context-free constrained path querying using RSM and the naive computation strategy.

Suppose that the input is a graph D presented in figure 2 and the grammar G has two productions $S \to a\ b \mid a\ S\ b$. The start vertex is v_0 and our goal is to find at least one path to each reachable vertex (w.r.t. G). The extended RSM for the given grammar is presented in figure 1.

The initial configuration is $(q_4, v_0, [])$: we start from the initial state of the box for S', the initial position in the graph is a start vertex v_0, the stack is empty. In each step, we apply rules from definition 10 to compute new configurations. The sequence of transitions, presented below, allows us to find a path

$$v_0 \xrightarrow{a} v_0 \xrightarrow{b} v_1$$

from v_0 to v_1 (step 5) and a path

$$v_0 \xrightarrow{a} v_0 \xrightarrow{a} v_0 \xrightarrow{b} v_1 \xrightarrow{b} v_0$$

from v_0 to itself (step 9).

$$
\begin{aligned}
(q_4, v_0, []) &\vdash \{(q_0, v_0, [q_5])\} \qquad (1)\\
(q_0, v_0, [q_5]) &\vdash \{(q_1, v_0, [a, q_5])\} \qquad (2)\\
(q_1, v_0, [a, q_5]) &\vdash \{(q_0, v_0, [q_2, a, q_5])\\
&\quad , (q_3, v_1, [b, a, q_5])\} \qquad (3)\\
(q_0, v_0, [q_2, a, q_5]) &\vdash \{(q_1, v_0, [a, q_2, a, q_5])\} \qquad (4)\\
(q_3, v_1, [b, a, q_5]) &\vdash \boxed{\{(q_5, v_1, [S(a, b)])\}} \qquad (5)\\
(q_1, v_0, [a, q_2, a, q_5]) &\vdash \{(q_3, v_1, [b, a, q_2, a, q_5])\\
&\quad , (q_0, v_0, [q_2, a, q_2, a, q_5])\} \qquad (6)\\
(q_3, v_1, [b, a, q_2, a, q_5]) &\vdash \{(q_2, v_1, [S(a, b), a, q_5])\} \qquad (7)\\
(q_2, v_1, [S(a, b), a, q_5]) &\vdash \{(q_3, v_0, [b, S(a, b), a, q_5])\} \qquad (8)\\
(q_3, v_0, [b, S(a, b), a, q_5]) &\vdash \boxed{\{(q_5, v_0, [S(a, S(a, b), b)])\}} \qquad (9)
\end{aligned}
$$

Note that we have no conditions to stop computation. In our example, we can continue computation and get new paths between v_0 and v_1. Moreover, there is an infinite number of such paths. Additionally, the selection of the next step is not deterministic. One can choose the configuration $(q_0, v_0, [q_2, a, q_2, a, q_5])$ to continue computations after step 6 with a chance to fall into an infinite cycle, but choosing $(q_3, v_1, [b, a, q_2, a, q_5])$ we get a new path.

4. GENERALIZED LL FOR CFPQ

Our solution is based on the generalized LL parsing algorithm [20] which was shown to generalize well to graph processing [15]. As a result of parsing, GLL can construct a *Shared Packed Parse Forest* (SPPF) [21] — a special data structure which represents all possible derivations of the input in a compressed form. Alternatively, GLL can provide the result in the form of *binary subtree sets* [22]. It was shown in [15] that SPPF can be naturally used to represent the result of the query for the *all-paths* problem finitely (see 3 and 5 in the set of problems).

In our work, we modify the GLL-based CFPQ algorithm to handle RSMs instead of grammars in BNF as a query specification. It enables performance improvement and native handling of both context-free and regular languages.

4.1. Paths Representation

Firstly, we need a way to represent a possibly infinite set of paths of interest. Note that the solution of the all-paths problem is a set of paths which can be treated as a language using the ω function (1). This language is a context-free language because it is an intersection of a context-free and a regular language. Thus, the paths can be represented in a way a context-free language can be represented.

The obvious way to represent a context-free language is to specify a respective context-free grammar. This way was used by Jelle Hellings in [12]: the query result is represented as a context-free grammar, and paths are extracted by generating strings.

Another way is a so-called *path index*: a data structure which is constructed during query processing and allows one to reconstruct paths of interest with the additional traversal. This idea is used in a wide range of graph analysis algorithms, notably by Rustam Azimov et al in [13]. In our case, the path index for the graph $D = \langle V, E, L\rangle$ and the query $\mathcal{R} = \langle \mathcal{N}, \Sigma, B, B_S, Q, Q_S\rangle$ is a $\mathcal{K} \times \mathcal{K}$ square matrix $\mathcal{I}$, where $\mathcal{K} = |Q| \cdot |V|$. Columns and rows are indexed by tuples of the form $(q_i, v_j), q_i \in Q, v_j \in V$. Each cell $\mathcal{I}[(q_i, v_j), (q_l, v_k)]$ represents information about a *range* $R^{q_i,v_j}_{q_l,v_k}$. *Range* or a *matched range* $R^{q_i,v_i}_{q_j,v_j}$ corresponds to the fact that there is a chain of transitions from the configuration $(q_i, v_i, \mathcal{S}_i)$ to the configuration $(q_j, v_j, \mathcal{S}_j)$, or $(q_i, v_i, \mathcal{S}_i) \vdash^* (q_j, v_j, \mathcal{S}_j)$. The symbol ϵ denotes the empty range: an empty sequence of configurations. Namely, $\mathcal{I}[(q_i, v_j), (q_l, v_k)]$ is a set which can contain the elements of three types:

- $t \in \Sigma$ means that $(q_i, v_j, \mathcal{S}') \vdash (q_l, v_k, t :: \mathcal{S}')$;
- $N_m \in \mathcal{N}$ means that $(q_i, v_j, \mathcal{S}') \vdash^* (q_l, v_k, N_m :: \mathcal{S}')$;
- I_{q_a,v_b} is an *intermediate point* that is used to denote that the range $R^{q_i,v_j}_{q_l,v_k}$ was combined from two ranges $R^{q_i,v_j}_{q_a,v_b}$ and $R^{q_a,v_b}_{q_l,v_k}$. Equivalently,
$$(q_i, v_j, \mathcal{S}') \vdash^* (q_a, v_b, \mathcal{S}'') \vdash^* (q_l, v_k, \mathcal{S}''').$$

A Shared Packed Parse Forest (SPPF) was proposed by Jan Rekers in [23] to represent parse forest without duplication of subtrees. Later, other versions of SPPF were introduced in different generalized parsing algorithms. It was also shown that SPPF is a natural way to compactly represent the structure of all paths that satisfy a CFPQ. In our work, we use SPPF with the following types of nodes.

- A *Terminal node* to represent a matched edge label.
- A *Nonterminal node* to represent the root of the subtree which corresponds to paths can be derived from the respective nonterminal.
- An *Intermediate node* which corresponds to the intermediate point used in the path index. This node has two children, both are range nodes.
- A *Range node* which corresponds to a matched range. It represents all possible ways to get a specific range and can have arbitrary many children. A child node can be of any type, besides a Range node. Nodes of this type can be reused.
- An *Epsilon node* to represent the empty subtree in the case when nonterminal produces the empty string.

In this paper, we describe a version that creates the path index, and then we show how to reconstruct SPPF using this index. The version provided can be easily adapted to build an SPPF directly during query evaluation, without creating the path index.

4.2. GLL-Based CFPQ Algorithm

In this section, we provide the detailed description of the GLL-based CFPQ algorithm that solves the *multiple source all-paths* problem and handles queries in the form of RSM. We use the improved version of GLL, proposed by Afroozeh and Izmaylova in [16], as a basis for our solution.

Firstly, we introduce the basic components of the algorithm.

Definition 12. *A descriptor is a 4-tuple* (q, v, s, r) *where:*

- *q is a state of RSM,*
- *v is a position in the input graph,*
- *s is a pointer to the current top of the stack,*
- *r is a matched range.*

A descriptor represents the state of the system completely; thus computation can be continued from the specified point without any additional information, only given a descriptor. There are two collections of descriptors in the algorithm. The first one is a collection $\mathcal{Q}$ of descriptors to handle. Each descriptor is handled once, so the second one is a collection of already handled descriptors $\mathcal{Q}_{\text{handled}}$.

Graph-structured stack (GSS) represents a set of stacks in a compact graph-like form which enables reusing of the common parts of stacks. We use an optimized version proposed by Afroozeh and Izmaylova in [16]. In this version, vertices identify calls, and edges contain return addresses and information about the matched part of the input. We slightly adapt GSS as follows:

Definition 13. *Suppose one has the graph $D = \langle V, E, L\rangle$ and evaluates the query $\mathcal{R} = \langle \mathcal{N}, \Sigma, B, B_S, Q, Q_S\rangle$ over it. Then a Graph Structured Stack (GSS) is a directed edge-labelled graph $D_{GSS} = \langle V_{GSS}, E_{GSS}, L_{GSS}\rangle$, where:*

- $V_{GSS} \subseteq Q \times V$,
- $E_{GSS} \subseteq V_{GSS} \times L_{GSS} \times V_{GSS}$, $L_{GSS} \subseteq Q \times R$,
- $R \subseteq Q \times V \times Q \times V$ *is a set of matched ranges.*

Two main operations over GSS are *push* and *pop*. Push adds a vertex and an edge if they do not already exist. Doing pop, we go through all outgoing edges to get return addresses and to get the new stack head. There is no predefined order to handle descriptors. Thus, the new outgoing edge can be added at any time. To guarantee that all possible outgoing edges will be handled, we should save information about what pops have been done. In our case, we save the matched range of the current descriptor. When new outgoing edges are added to the GSS vertex, we should check whether this vertex has been popped and handle the newly added outgoing edges if necessary.

We suppose that the input of the algorithm is an extended RSM $\mathcal{R}' = \langle \mathcal{N}', \Sigma', B', B'_S, Q', Q'_S\rangle$ and graph $D = \langle V, E, L\rangle$. Also note that we solve multiple-source CFPQ, so the set of start vertices $V_S \subseteq V$ is specified by the user. The first step of the algorithm is the initialization. For each start vertex $v_i \in V_S$, we create a GSS vertex $s = (q'_0, v_i)$ and then create a descriptor (q'_0, v_i, s, ϵ), add it to $\mathcal{Q}$. Here q'_0 is the initial state of B'_S.

In the main loop, we handle descriptors one by one. Namely, while $\mathcal{Q}$ is not empty, we pick a descriptor $d = (q_0, v_0, s_0, r_0), r_0 = R^{p_0,u_0}_{q_0,v_0}$, add it to $\mathcal{Q}_{\text{handled}}$, and process it. Processing consists of the following cases that provide a way to compute the transition step of RSM, as defined in 10, extended with path index creation.

Let Δ be all possible transitions from all boxes:

$$\Delta = \bigcup_{\substack{B_k \in B' \\ B_k = \langle Q^k, q^k_S, Q^k_F, \delta^k\rangle}} \delta^k.$$

1. **Handling of terminal transitions.** The terminal transitions are handled similarly to transitions in a finite automaton: for all matched terminals, we simultaneously move forward to the next state and the next vertex; the stack does not change; the matched range extends right with the matched symbol. Thus, the new set of descriptors is
$$\begin{aligned} D_1 = \{&(q_1, v_1, s_0, R^{p_0,u_0}_{q_1,v_1}) \\ &\mid (v_0, t, v_1) \in E, (q_0, t, q_1) \in \Delta, t \in \Sigma\}. \end{aligned} \tag{10}$$
This time, we add the terminal t to $\mathcal{I}[(q_0, v_0), (q_1, v_1)]$ and the intermediate point I_{q_0,v_0} to $\mathcal{I}[(p_0, u_0), (q_1, v_1)]$.

2. **Handling of nonterminal transitions.** For each nonterminal symbol N_0 such that there exists a transition (q_0, N_0, q_1), we should start processing of the nonterminal N_0. To achieve it, we should push the return address q_1 and the accumulated range r_0 to the stack. To do it, the vertex s_1 with label (q_0, v_0) is created or reused. Next, we create the descriptor $d = (q^{N_0}_0, v_0, s_1, \epsilon)$, where $q^{N_0}_0$ is a start state of B_{N_0}. Additionally, we should handle the stored pops not to miss the newly added outgoing edge. For all stored matched ranges $R^{q_0,v_0}_{q_3,v_3}$ for the current GSS vertex s_1, we create the range $R^{q_0,v_0}_{q_1,v_3}$. Finally, we combine the new range $R^{p_0,u_0}_{q_1,v_3}$ from the range of the current descriptor, the newly created one, and the respective

descriptor $(q_0, v_3, q', R^{p_0,u_0}_{q_1,v_3})$. We also add intermediate points of created descriptors to $\mathcal{I}$. Finally,

$$\begin{aligned} D_2 = \{ & q_0^{N_0}, v_0, s_1, \epsilon \\ & \mid s_1 = (q_0, v_0), (q_0, N_0, q_1) \in \Delta, \\ & q_0^{N_0} \text{ is a start state of } B_{N_0}\} \\ \cup\{ & q_1, v_3, s_0, R^{p_0,u_0}_{q_1,v_3} \\ & \mid s_1 = (q_0, v_0), (q_0, N_0, q_1) \in \Delta, \\ & R^{q_0,v_0}_{q_3,v_3} \in s_1.\text{stored\_pops}\}. \end{aligned} \tag{11}$$

3. **Handling of the final state**. If q_0 is a final state of B_{N_0}, it means that the recognition of the corresponding nonterminininal is completed. So, we should pop from the stack to determine the point from which we restart the recognition and return the respective states to continue. Namely, for all outgoing edges from $s_0 = (r_0, w_0)$ of form $(s_0, (q_1, R^{q_2,v_2}_{q_3,w_0}), s_1)$ we use q_1 as the return address and use the range from label to create the following ranges. First of all, we create the range $R^{q_3,w_0}_{q_1,v_0}$ that corresponds to the recognized nonterminal, and save N_0 to $\mathcal{I}[(q_3, w_0), (q_1, v_0)]$. Next, we combine this range with the range from GSS edge getting $R^{q_2,v_2}_{q_1,v_0}$, and store intermediate point I_{q_3,w_0} to $\mathcal{I}[(q_2, v_2), (q_1, v_0)]$. Note, that we should store the information about these pops. Finally,

$$\begin{aligned} D_3 = \{ & (q_1, v_0, s_1, R^{q_2,v_2}_{q_1,v_0}) \\ & \mid (s_0, (q_1, R^{q_2,v_2}_{q_3,w_0}), s_1) \in E_{\text{GSS}}, s_0 = (r_0, w_0)\}. \end{aligned} \tag{12}$$

Now, we have a set of the newly created descriptors $D_{\text{created}} = D_1 \cup D_2 \cup D_3$. We drop out the descriptors which have been already handled, add the rest to $\mathcal{Q}$, and repeat the main loop again.

The result of the algorithm is a path index $\mathcal{I}$, which allows one to check reachability or to reconstruct paths or SPPF by traversing starting from the cell corresponding to the range of interest.

4.3. Example

Suppose that we have the same graph and RSM, as we have used in 3.3.1 (2 and 1 respectively). We construct a path index and then show how to create SPPF using it. All the steps of the index creation algorithm are presented in the table 1. The final index is presented in figure 4. Note that this example is rather simple, so we avoid handling the stored pops.

The first step is the initialization. The stack is initialized with one vertex which represents the initial call, while the range is empty.

In the second step, we start to handle the new nonterminal. So, the new vertex and the respective edge are added to GSS.

In the third step, we match the terminal a and create the first record in the path index.

In the fourth step, we produce two descriptors. The first one has already been handled, so we should not process it again (thus, it is crossed out), but the creation of this descriptor leads to a new call. So, the new vertex and edge are added to GSS. Note that the respective vertex already exists, so only the new edge is added. Stack will not change any more after this step.

In the fifth step, two descriptors are created. One of them (boxed) denotes that we have recognized a path of interest. The second unique path is finished at the seventh step. All other paths are a combination of these two.

At the sixth step, the terminal b from the edge $v_1 \xrightarrow{b} v_0$ is handled.

Finally, at the eighth step, we get the empty set of descriptors to process $\mathcal{Q}$ and finish the process.

$$\left(\begin{array}{cccccccccccccccc}
 & q_0,v_0 & q_1,v_0 & q_2,v_0 & q_3,v_0 & q_4,v_0 & q_5,v_0 & q_6,v_0 & q_0,v_1 & q_1,v_1 & q_2,v_1 & q_3,v_1 & q_4,v_1 & q_5,v_1 & q_6,v_1 & \\
q_0,v_0 & & \{a\} & \{I_{q_1,v_0}\} & \{I_{q_2,v_1}\} & & & & & & \{I_{q_1,v_0}\} & \{I_{q_1,v_0}, I_{q_2,v_0}\} & & & & q_0,v_0 \\
q_1,v_0 & & & \{S\} & & & & & & & \{S\} & \{b\} & & & & q_1,v_0 \\
q_2,v_0 & & & & & & & & & & & \{b\} & & & & q_2,v_0 \\
q_3,v_0 & & & & & & & & & & & & & & & q_3,v_0 \\
q_4,v_0 & & & & & & \{S\} & & & & & & & \{S\} & & q_4,v_0 \\
q_5,v_0 & & & & & & & & & & & & & & & q_5,v_0 \\
q_6,v_0 & & & & & & & & & & & & & & & q_6,v_0 \\
q_0,v_1 & & & & & & & & & & & & & & & q_0,v_1 \\
q_1,v_1 & & & & & & & & & & & & & & & q_1,v_1 \\
q_2,v_1 & & & & \{b\} & & & & & & & & & & & q_2,v_1 \\
q_3,v_1 & & & & & & & & & & & & & & & q_3,v_1 \\
q_4,v_1 & & & & & & & & & & & & & & & q_4,v_1 \\
q_5,v_1 & & & & & & & & & & & & & & & q_5,v_1 \\
q_6,v_1 & & & & & & & & & & & & & & & q_6,v_1 \\
 & q_0,v_0 & q_1,v_0 & q_2,v_0 & q_3,v_0 & q_4,v_0 & q_5,v_0 & q_6,v_0 & q_0,v_1 & q_1,v_1 & q_2,v_1 & q_3,v_1 & q_4,v_1 & q_5,v_1 & q_6,v_1 &
\end{array}\right)$$

Figure 4. The path index created during the steps represented in table 1.

Next, we show how to reconstruct SPPF from path index. Suppose we want to build SPPF for all paths that start in v_0, finish in v_0, and form words derivable from S. To achieve this, we start from the respective root range node $R^{q_4,v_0}_{q_5,v_0}$. The information about ways to build this range is stored in $\mathcal{I}[(q_4,v_0),(q_5,v_0)]$. Each element of the respective set becomes a child of the respective range node. In our case, we see that this range corresponds to the nonterminal node S. Thus, we should find all the ways to build range $R^{q_0,v_0}_{q_3,v_0}$ (q_0 is a start state for S and q_3 is a final state). To do this, we look at $\mathcal{I}[(q_0,v_0),(q_3,v_0)]$. It contains the information about the intermediate point I_{q_2,v_1} which means range $R^{q_0,v_0}_{q_3,v_0}$ has been built from the two ranges $R^{q_0,v_0}_{q_2,v_1}$ and $R^{q_2,v_1}_{q_3,v_0}$ that became children nodes of I_{q_2,v_1}. Repeating this procedure allows us to build SPPF in a top-down direction. At some steps, for example, when processing the intermediate point I_{q_1,v_0}, we found that some range nodes are always added to SPPF. Such nodes should be reused. The final result is presented in figure 3.

Note that one can extract paths explicitly instead of SPPF construction in a similar fashion.

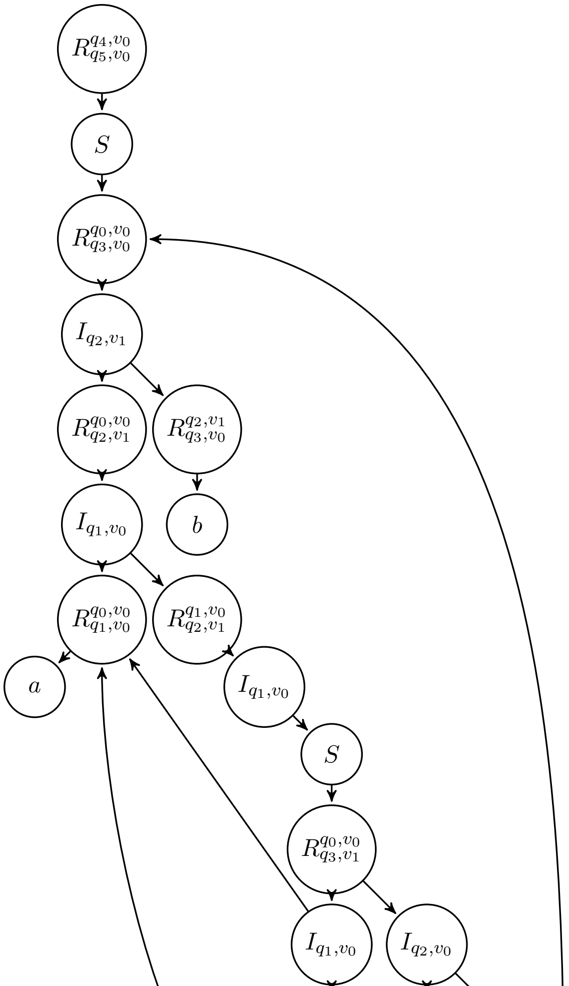
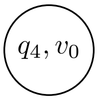
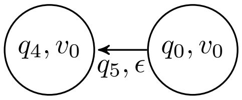
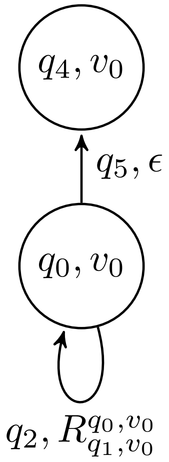


Table 1. The steps of the GLL-based CFPQ executed on the graph represented in figure 2 and the query represented in figure 1.

	Configuration	Path	Stack
1	$\vdash (q_4, v_0, (q_4, v_0), \epsilon)$		
2	$(q_4, v_0, (q_4, v_0), \epsilon) \vdash \{(q_0, v_0, (q_0, v_0), \epsilon)\}$		
3	$(q_0, v_0, (q_0, v_0), \epsilon) \vdash \{(q_1, v_0, (q_0, v_0), R^{q_0,v_0}_{q_1,v_0})\}$	$[(q_0, v_0), (q_1, v_0)] \leftarrow a$	
4	$(q_1, v_0, (q_0, v_0), R^{q_0,v_0}_{q_1,v_0}) \vdash$ $\{\cancel{(q_0, v_0, (q_0, v_0), \epsilon)}, (q_3, v_1, (q_0, v_0), R^{q_0,v_0}_{q_3,v_1})\}$	$[(q_1, v_0), (q_3, v_1)] \leftarrow b$ $[(q_0, v_0), (q_3, v_1)] \leftarrow (q_1, v_0)$	 q_4, v_0 q_5, ϵ q_0, v_0 $q_2, R^{q_0,v_0}_{q_1,v_0}$
5	$(q_3, v_1, (q_0, v_0), R^{q_0,v_0}_{q_3,v_1}) \vdash$ $\{(q_2, v_1, (q_0, v_0), R^{q_0,v_0}_{q_2,v_1}), \boxed{(q_5, v_1, (q_4, v_0), R^{q_4,v_0}_{q_5,v_1})}\}$	$[(q_4, v_0), (q_5, v_1)] \leftarrow S$ $[(q_1, v_0), (q_2, v_1)] \leftarrow S$ $[(q_0, v_0), (q_2, v_1)] \leftarrow (q_1, v_0)$	
6	$(q_2, v_1, (q_0, v_0), R^{q_0,v_0}_{q_2,v_1}) \vdash \{(q_3, v_0, (q_0, v_0), R^{q_0,v_0}_{q_3,v_0})\}$	$[(q_2, v_1), (q_3, v_0)] \leftarrow b$ $[(q_0, v_0), (q_3, v_0)] \leftarrow (q_2, v_1)$	
7	$(q_3, v_0, (q_0, v_0), R^{q_0,v_0}_{q_3,v_0}) \vdash$ $\{(\mathrm{q}_2, v_0, (q_0, v_0), R^{q_0,v_0}_{q_2,v_0}), \boxed{(q_5, v_0, (q_4, v_0), R^{q_4,v_0}_{q_5,v_0})}\}$	$[(q_4, v_0), (q_5, v_0)] \leftarrow S$ $[(q_1, v_0), (q_2, v_0)] \leftarrow S$ $[(q_0, v_0), (q_2, v_0)] \leftarrow (q_1, v_0)$	
8	$(q_2, v_0, (q_0, v_0), R^{q_0,v_0}_{q_2,v_0}) \vdash \{\cancel{(q_3, v_1, (q_0, v_0), R^{q_0,v_0}_{q_3,v_1})}\}$	$[(q_0, v_0), (q_3, v_1)] \leftarrow (q_2, v_0)$ $[(q_2, v_0), (q_3, v_1)] \leftarrow b$	

5. EXPERIMENT DESIGN

In this section, we provide a description of graphs and queries used for the evaluation of implemented algorithms. Also, we describe common evaluation scenarios and the evaluation environment.

Table 2. RDF graphs for evaluation: number of vertices and edges, and number of edges with specific label.

Graph name	$\lvert V\rvert$	$\lvert E\rvert$	#subClassOf	#type	#bt
Core	1 323	2 752	178	0	0
Pathways	6 238	12 363	3 117	3 118	0
Go_hierarchy	45 007	490 109	490 109	0	0
Enzyme	48 815	86 543	8 163	14 989	8 156
Eclass	239 111	360 248	90 962	72 517	0
Geospecies	450 609	2 201 532	0	89 065	20 867
Go	582 929	1 437 437	94 514	226 481	0
Taxonomy	5 728 398	14 922 125	2 112 637	2 508 635	0

We evaluated our solution on both classical regular and context-free path queries to estimate the ability to use the proposed algorithm as a universal algorithm for a wide range of queries.

5.1. Graphs

For the evaluation, we use a number of graphs from the CFPQ_Data [24] data set. We selected a number of graphs related to RDF analysis. A detailed description of the graphs, namely the number of vertices and edges and the number of edges labeled by tags used in queries, is in Table 2. Here "bt" is an abbreviation for *broaderTransitive* relationship.

For regular path queries evaluation, we used only RDF graphs because code analysis related graphs contain only two types of labels. Regular queries over such graphs are meaningless.

5.2. Regular Queries

Regular queries were generated using a well-established set of templates for RPQ algorithms evaluation. Namely, we use templates presented in Table 2 in [25] and in Table 5 in [26]. We select four non-trivial templates (that contain compositions of Kleene star and union) that are expressible in Cypher syntax to be able to compare the native Neo4j querying algorithm with our solution. Used templates are presented in equations 13, 14, 15, and 16. Respective path patterns expressed in Cypher are presented in equations 17, 18, 19, and 20. Note that while Cypher's power is limited, our solution can handle arbitrary RPQs. We generate one query for each template and each graph. The most frequent relations from the given graph were used as symbols in the query template.

$$\mathit{reg}_1 = (a \mid b)^* \tag{13}$$

$$\mathit{reg}_2 = a^* \cdot b^* \tag{14}$$

$$\mathit{reg}_3 = (a \mid b \mid c)^+ \tag{15}$$

$$\mathit{reg}_4 = (a \mid b)^+ \cdot (c \mid d)^+ \tag{16}$$

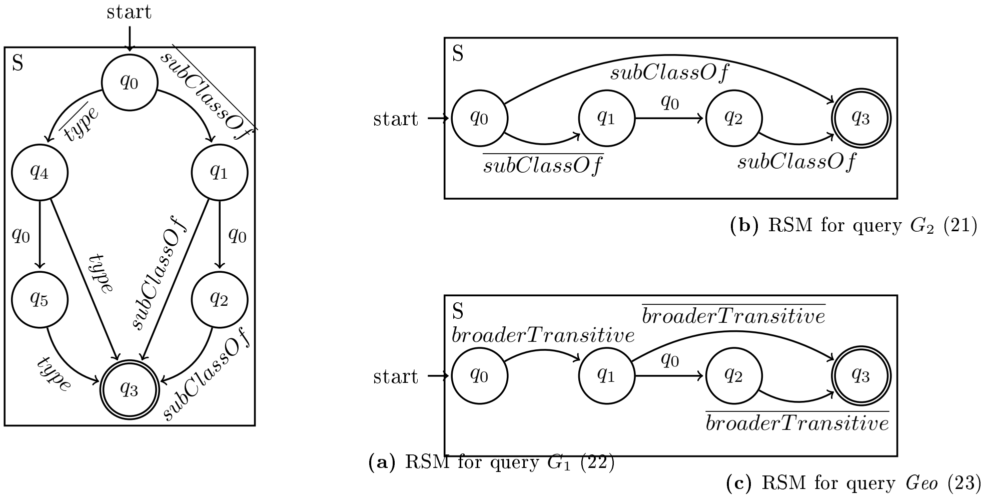
$$\mathit{reg}_1^{\mathrm{N4j}} = \texttt{()-[:a | :b]->\{0,\}()} \tag{17}$$

$$\mathit{reg}_2^{\mathrm{N4j}} = \texttt{()-[:a]->\{0,\}()-[:b]->\{0,\}()} \tag{18}$$

$$\mathit{reg}_3^{\mathrm{N4j}} = \texttt{()-[:a | :b | :c]->\{1,\}()} \tag{19}$$

$$\mathit{reg}_4^{\mathrm{N4j}} = \texttt{()-[:a | :b]->\{1,\}()-[:c | :d]->\{1,\}()} \tag{20}$$

Also note that exclude *go_hierarchy* graph from evaluation because it contains only one type of edge, so it is impossible to express meaningful queries over it.



(a) RSM for query G_1 (22)

(b) RSM for query G_2 (21)

(c) RSM for query *Geo* (23)

Figure 5. RSMs for queries.

5.3. Context-Free Queries

All queries used in our evaluation are variants of *same-generation query*. For the *RDF* graphs we use the same queries that were used for CFPQ algorithms evaluation in the other works [1, 2]: G_1 (22), G_2 (21), and *Geo* (23). The queries are expressed as context-free grammars where S is a nonterminal, *subClassOf, type, broaderTransitive,* $\overline{subClassOf}$, $\overline{type}$, $\overline{broaderTransitive}$ are terminals or the labels of edges. We denote the inverse of an x relation and the respective edge as $\overline{x}$.

$$S \to \overline{subClassOf}\ \ S\ subClasOf \mid subClassOf \tag{21}$$

$$\begin{aligned} S \to & \overline{subClassOf}\ \ S\ subClasOf \mid \overline{type}\ \ S\ type \\ & \mid \overline{subClassOf}\ \ subClassOf \mid \overline{type}\ type \end{aligned} \tag{22}$$

$$\begin{aligned} S \to & broaderTransitive\ \ S\ \overline{broaderTransitive} \\ & \mid broaderTransitive\ \ \overline{broaderTransitive} \end{aligned} \tag{23}$$

Respective RSMs are presented in figures 5a for G_1, 5b for G_2, and 5c for *Geo*.

5.4. Scenarios Description

We evaluate the proposed solution on the *multiple sources reachability* scenario. We assume that the size of the starting set is significantly less than the number of the input graph vertices. This limitation looks reasonable in practical cases.

The starting sets for the multiple sources querying are generated from all vertices of a graph with a random permutation. We use chunks of size 1, 10, 100. For graphs with less than 10 000 vertices, all vertices were used for querying. For graphs with from 10 000 to 100 000 vertices, 10% of vertices were considered starting ones. For the graphs with more than 100 000 vertices, only 1% of vertices were considered. We use the same sets for all cases in all experiments to be able to compare results.

To check the correctness of our solution and to force the result stream, we compute the number of reachable pairs for each query.

5.5. *Evaluation Environment*

We ran all experiments on a PC with Ubuntu 20.04 installed. It has an Intel Core i7-6700 CPU, 3.4GHz, 4 threads (hyper-threading is turned off), and DDR4 64Gb RAM. We use OpenJDK 64-Bit Server VM Corretto-17.0.8.8.1 (build 17.0.8.1+8-LTS, mixed mode, sharing). JVM was configured to use 55Gb of heap memory: both `xms` and `xmx` are set to 55Gb.

We use Neo4j 5.12.0. Almost all configurations of Neo4j are default. We only set `memory_transaction_global_max_size` to `0`, which means unlimited memory usage per transaction.

As a competitor for our implementation, we use a linear algebra-based solution, integrated to RedisGraph by Arseniy Terekhov et al. and described in [27] and we use the configuration described in it for RedisGraph evaluation in our work.

6. PERFORMANCE OF GLL ON QUERIES IN BNF AND EBNF

As discussed above, different ways to specify context-free grammars can be used to specify the query. The basic one is BNF (see definition 2), the more expressive (but not more powerful) is EBNF (see definition 6). EBNF is not only more expressive, but potentially allows one to improve the performance of query evaluation because it avoids stack usage by replacing some recursive rules with the Kleene star. RSMs allow one to natively represent grammars in EBNF and can be handled by GLL as described in section 4.

We implemented and evaluated two versions of the GLL-based CFPQ algorithm [28]: one operates with grammar in BNF, and another one operates with grammar in EBNF and utilizes RSM to represent it. At this step, we use simple data structures for graph and query representations, tuned for our algorithm. Both versions were evaluated in reachability-only mode to estimate performance differences and to choose the best one to integrate with Neo4j.

6.1. *Evaluation*

To assess the applicability of the proposed solution, we evaluate it on a number of real-world graphs and queries described in section 5.

We compare the performance of evaluation of queries in *reachability-only* mode with different sizes of the start vertex set to estimate the speedup of the RSM-based version relative to the BNF-based one.

The experimental study was conducted as follows.

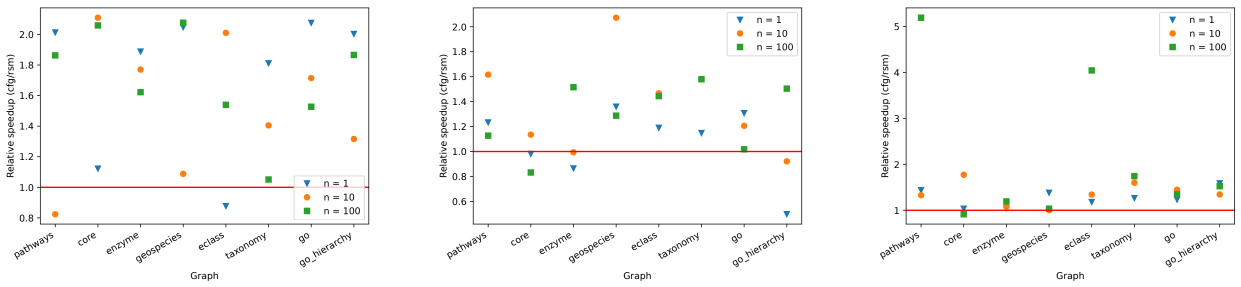
- For all graphs, queries and start vertices sets, that described in section 5, we measure evaluation time for both versions.
- Average time for each start vertices set size was calculated. So, for each graph, query, and start vertices set size we have an average time of respective query evaluation.
- Speedup as a ratio of BNF-based version evaluation time to RSM-based version evaluation time was calculated.

Results presented in figures 6a, 6b, and 6c for queries G_1, G_2 and *Geo* respectively. We can see that in almost all cases the RSM-based version is faster than the BNF-based one. While in most cases speedup is not greater than 2, in some cases it can be more than 5 (see figure 6c: graph *pathways*, grammar *Geo*). Average speedup over all graphs and grammars is 1.5. So we can conclude that RSM-based GLL demonstrates better performance than the BNF-based one on average.

7. CFPQ FOR NEO4J

In this section we provide details on the integration of GLL-based CFPQ to Neo4j graph database. We choose the RSM-based version because our comparison (see section 6.6.1) shows that it is faster than the BNF-based one.

Also, we provide results of the implemented solution evaluation which show that, first of all, the provided solution is faster than a similar linear algebra-based solution for RedisGraph. Also, we show that on RPQs our solution is compatible with the Neo4j native one and in some cases evaluates queries successfully while the native solution fails with an *OutOfMemory* exception.



(a) G_1 query

(b) G_2 query

(c) *Geo*

Figure 6. Multiple sources CFPQ reachability speedup (RSM over CFG) on RDF graphs.

7.1. Implementation Details

Neo4j stored procedure is a mechanism through which query language can be extended by writing custom code in Java in such a way that it can be called directly via Cypher.

We implemented a Neo4j stored procedure which solves the reachability problem for the given set of start vertices and the given query. The procedure can be called as follows: `CALL cfpq.gll.getReachabilities(nodes, q)` where `nodes` is a collection of start nodes, and `q` is a string representation (or description) of RSM specified over relation types. The result of the given procedure is a stream of reachable pairs of nodes. Note that the expressive power of our solution is limited: we cannot use the full power of Cypher inside our constraints. For example, we cannot specify constraints on vertices inside our constraints.

We implemented a wrapper for Neo4j. Communication with the database is done using the Neo4j Native Java API. So, we used an embedded database, which means it is run inside of the application and is not used as an external service.

Along with the existing modifications of GLL, we made a Neo4j-specific one. Neo4j return result should be represented as a `Stream` and it is important to prevent early stream forcing, thus we changed all GLL internals to ensure that. This also has an added benefit that the query result is a stream, and thus it is possible to get the results on demand.

7.2. Evaluation

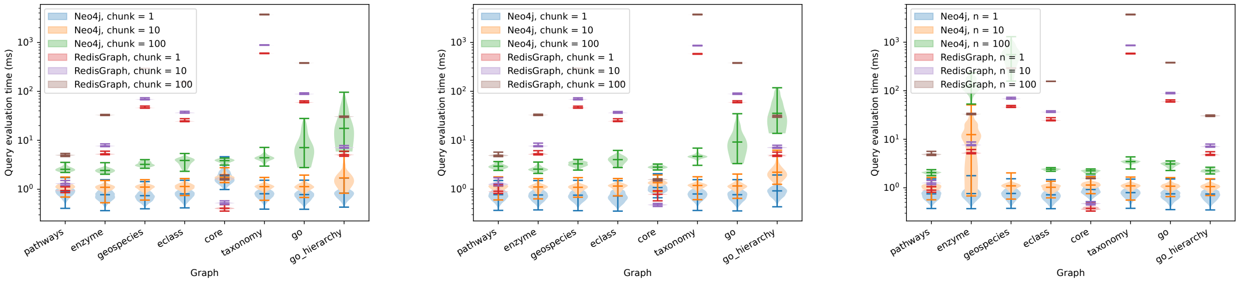
To assess the applicability of the proposed solution, we evaluate it on a number of real-world graphs and queries. To estimate relative performance, we compare our solution with the matrix-based CFPQ algorithm implemented in RedisGraph by Arseniy Terekhov et al in [27]. Also, we compare the performance of query evaluation in *reachability-only* mode on regular path queries with the native Neo4j solution.

The results of context-free path query evaluations are presented in figures 7a for G_1, 7b for G_2, and 7c for *Geo*.

The results show that query evaluation time depends not only on a graph size or its sparsity, but also on the inner structure of the graph. For example, the relatively small graph *go_ hierarchy* fully consists of edges used in queries G_1 and G_2, thus evaluation time for these queries is significantly bigger than for some bigger but more sparse graphs, for example, for the *eclass* graph. Especially for a relatively big starting vertex set. Note that the creation of relevant metrics for CFPQ query evaluation time prediction is a challenging problem by itself and should be tackled in the future.

Also, we can see that in almost all cases the proposed solution is significantly faster than RedisGraph (in orders of magnitude in some cases). At the same time, in some cases (see results for graph *core* and all queries) RedisGraph outperforms our solution. Also, one can see that evaluation time for RedisGraph is more predictable. For our solution, in some cases, execution time highly depends on the start set. For example, see figure 7c, graph *enzyme*.

The particularly important scenario is the case when the start set is a single vertex. The results of the `single-source reachability` show that such queries are reasonably fast: median time is



(a) G_1 query **(b)** G_2 query **(c)** *Geo*

Figure 7. Multiple sources CFPQ reachability results for queries related to RDF analysis.

about a few milliseconds for all graphs and all queries. Note that even for single source queries, evaluation time highly depends on graph structure: evaluation time on *core* graph is significantly bigger than for all other graphs for all queries. Note that *core* is the smallest graph in terms of the number of nodes and edges. Again, to provide a reliable metric to predict query execution time is a non-trivial task. Moreover, time grows with the size of a chunk, as expected.

Partial results for RPQ evaluation are presented in figures 8a and 8b for reg_1 (defined in 13) and reg_2 (defined in 14) respectively. For queries reg_3 (defined in 15) and reg_4 (defined in 16) we get similar results. Note that on *geospecies* and *taxonomy* graphs, the native solution failed with an *OutOfMemory* exception, while our solution evaluates queries successfully.

We can see that the proposed solution is slightly slower than the native Neo4j algorithm, but not dramatically: typically less than two times. Moreover, in some cases, our solution is comparable with the native one (see figure 8a and figure 8b, graph *eclass*), and in some cases, our solution is faster than the native one (see figure 8b, graph *core*).

Finally, we conclude that not only can the linear algebra-based CFPQ algorithms be used in real-world graph analysis systems: the proposed GLL-based solution outperforms the linear algebra-based one. Moreover, we show that the proposed solution can be used as a universal algorithm for both RPQ and CFPQ.

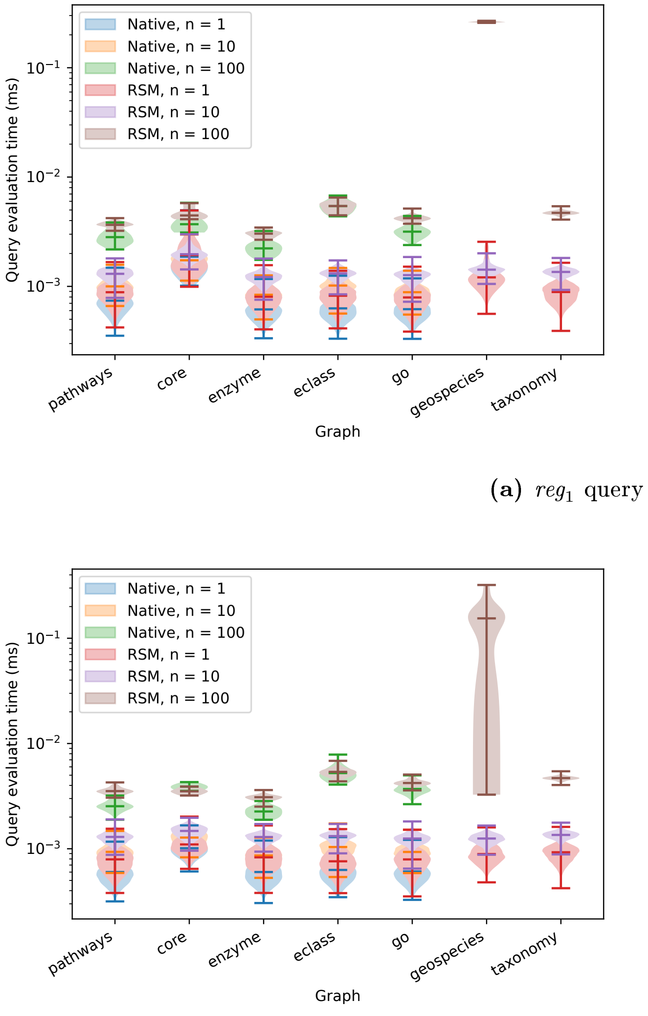
8. RELATED WORK

The idea to use context-free grammars as constraints in the path finding problem in graph databases was introduced and explored by Mihalis Yannakakis in [29]. A bit later, the same idea was developed by Thomas Reps et al. as a general framework for static code analysis [30]. Further, this framework, called Context-Free Language Reachability (CFL-r), became one of the most popular and actively developed. The landscape analysis of the area was recently provided by Andreas Pavlogiannis in [31]. In the context of graph databases, the most recent systematic comparison of different CFPQ algorithms was done by Jochem Kuijpers et al. A set of CFPQ algorithms was implemented and evaluated using Neo4J as a graph storage. Results were presented in [2]. It was concluded that the existing algorithms are not performant enough to be used for real-world data analysis.

Regarding graph databases, CFPQ was applied in such graph analysis related tasks as biological data analysis [4], data provenance [5], hierarchy analysis in RDF data [3, 32].

Multiple CFPQ algorithms are based on different parsing algorithms and techniques. For example, an approach based on parsing combinators was proposed by Ekaterina Verbitskaia et al. in [33]. Several algorithms based on LL-like and LR-like techniques were developed by Ciro M. Medeiros et al. in [32, 34, 35]. Also, CFPQ algorithms were investigated by Phillip Bradford in [36, 37] and Charles B. Ward et al. in [38]. An algorithm based on matrix equations was proposed by Yuliya Susanina in [39]. The paths extraction problem was studied by Jelle Hellings in [12].

A set of linear algebra-based CFPQ algorithms was developed, including all-paths and single-path versions, proposed by Rustam Azimov et al. in [13] and [11] respectively. The Kronecker



(a) *reg_1* query

(b) *reg_2* query

Figure 8. Multiple source RPQ reachability results for queries related to RDF analysis and respective query (native solution failed with OOM on last two graphs).

product-based algorithm [40] was proposed by Egor Orachev et al., and a multiple-source CFPQ algorithm for RedisGraph was proposed by Arseniy Terekhov et al. in [27].

Recursive state machines were studied in the context of CFPQ in several papers, including [40] where Egor Orachev et al. use RSM to specify context-free constraints, Yuxiang Lei et al. [41] propose to use RSM to specify path constraints, and [42] where Swarat Chaudhuri proposes a slightly subcubic algorithm for the reachability problem for recursive state machines — the equivalent to the CFPQ problem.

GLL was introduced by Elizabeth Scott and Adrian Johnstone in [20]. A number of modifications of the GLL algorithm were further proposed, including the version which supports EBNF without its transformation [43] and the version which uses binary subtree sets [22] instead of SPPF. The latest version simplifies the algorithm and avoids the overhead of explicit graph construction. Within it, the optimized and simplified OOP-friendly version of GLL was proposed by Ali Afroozeh and Anastasia Izmaylova in [16].

9. CONCLUSION AND FUTURE WORK

In this work, we presented the GLL-based context-free path querying algorithm for the Neo4j graph database. The implementation is available on GitHub: `https://github.com/vadyushkins/cfpq-gll-neo4j-procedure`.

Our solution uses Neo4j for graph storage, but the query language should be extended to support context-free constraints to make it useful. Both the extension of Cypher and the integration of our algorithm with the query engine are non-trivial challenges left for future work.

While GLL-based CFPQ potentially can be used to solve *all-paths* problem, currently we implemented the procedure for the reachability problem only. Choosing useful strategies to enumerate paths and implementation of them is a direction for future research.

The most important direction for future work is to find a way to provide an incremental version of the GLL-based CFPQ algorithm to avoid full query reevaluation when the graph is only slightly changed. While our solution is based on the well-known parsing algorithm and there are solutions for incremental parsing, development of the efficient incremental version of the GLL-based CFPQ algorithm is a challenging problem left for future research.

Another direction is to create a parallel version of the GLL-based CFPQ algorithm to improve its performance on huge graphs. Although it seems natural to handle descriptors in parallel, the algorithm operates over global structures, and the naive implementation of this idea leads to a significant overhead in synchronization.

9.0.1. We thank Adrian Johnstone for pointing out the Generalized LL algorithm in our discussion at Parsing@SLE–2013, which motivated the development of the presented solution. We thank George Fletcher for the discussion of the evaluation of different CFPQ algorithms for Neo4j.

9.0.2. The authors have no competing interests to declare that are relevant to the content of this article.

REFERENCES

[1] R. Azimov and S. Grigorev, 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, 2018) pp. 5:1–5:10.

[2] J. Kuijpers, G. Fletcher, N. Yakovets, and T. Lindaaker, in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, SSDBM '19 (ACM, New York, NY, USA, 2019) pp. 121–132.

[3] X. Zhang, Z. Feng, X. Wang, G. Rao, and W. Wu, in *The Semantic Web – ISWC 2016*, edited by P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil (Springer International Publishing, Cham, 2016) pp. 632–648.

[4] P. Sevon and L. Eronen, Journal of Integrative Bioinformatics **5**, 157 (2008).

[5] H. Miao and A. Deshpande, in *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019) pp. 1710–1713.

[6] X. Zheng and R. Rugina, in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08 (ACM, New York, NY, USA, 2008) pp. 197–208.

[7] J. Rehof and M. Fähndrich, SIGPLAN Not. **36**, 54 (2001).

[8] F. C. Santos, U. S. Costa, and M. A. Musicante, in *Web Engineering*, edited by T. Mikkonen, R. Klamma, and J. Hernández (Springer International Publishing, Cham, 2018) pp. 225–233.

[9] J. Hellings, in *Proceedings of ICDT'14* (2014) pp. 119–130.

[10] C. M. Medeiros, M. A. Musicante, and U. S. Costa, in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18 (Association for Computing Machinery, New York, NY, USA, 2018) p. 1230вЂ“1237.

[11] A. Terekhov, A. Khoroshev, R. Azimov, and S. Grigorev, 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, 2020).

[12] J. Hellings, in *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics*, edited by L. Qin, W. Zhang, Y. Zhang, Y. Peng, H. Kato, W. Wang, and C. Xiao (Springer International Publishing, Cham, 2020) pp. 84–98.

[13] R. Azimov, I. Epelbaum, and S. Grigorev, in *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '21 (Association for Computing Machinery, New York, NY, USA, 2021).

[14] O. Rodriguez-Prieto, A. Mycroft, and F. Ortin, IEEE Access **8**, 72239 (2020).

[15] S. Grigorev and A. Ragozina, in *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '17 (ACM, New York, NY, USA, 2017) pp. 10:1–10:7.

[16] A. Afroozeh and A. Izmaylova, in *Compiler Construction*, edited by B. Franke (Springer Berlin Heidelberg, Berlin, Heidelberg, 2015) pp. 89–108.

[17] Iguana parsing framework, `https://iguana-parser.github.io/`, accessed: 12.11.2024.

[18] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis, ACM Trans. Program. Lang. Syst. **27**, 786вЂ“818 (2005).

[19] Y. Bar-Hillel, M. Perles, and E. Shamir, Z. Phonetik Sprachwiss. Kommunikat. **14**, 143 (1961).

[20] E. Scott and A. Johnstone, Electronic Notes in Theoretical Computer Science **253**, 177 (2010), proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[21] E. Scott and A. Johnstone, Science of Computer Programming **78**, 1828 (2013), special section on Language Descriptions Tools and Applications (LDTAвЂ™08 & вЂ™09) & Special section on Software Engineering Aspects of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011).

[22] E. Scott, A. Johnstone, and L. T. van Binsbergen, Science of Computer Programming **175**, 63 (2019).

[23] J. G. Rekers, *Parser generation for interactive environments*, Ph.D. thesis, Citeseer (1992).

[24] Cfpq_data: A public set of graphs and grammars for cfpq algorithms evaluation, `https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_Data`, accessed: 27.09.2024.

[25] A. Pacaci, A. Bonifati, and M. T. Özsu, in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20 (Association for Computing Machinery, New York, NY, USA, 2020) pp. 1415–1430.

[26] X. Wang, S. Wang, Y. Xin, Y. Yang, J. Li, and X. Wang, World Wide Web **23**, 1465 (2019).

[27] A. Terekhov, V. Pogozhelskaya, V. Abzalov, T. Zinnatulin, and S. V. Grigorev, in *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, edited by Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, and F. Guerra (OpenProceedings.org, 2021) pp. 487–492.

[28] Gll-based cfpq algorithm implementation, `https://github.com/vadyushkins/kotgll`, accessed: 12.11.2024.

[29] M. Yannakakis, in *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90 (Association for Computing Machinery, New York, NY, USA, 1990) p. 230вЂ“242.

[30] T. Reps, S. Horwitz, and M. Sagiv, in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95 (Association for Computing Machinery, New York, NY, USA, 1995) p. 49вЂ“61.

[31] A. Pavlogiannis, ACM SIGLOG News **9**, 5вЂ“25 (2023).

[32] C. M. Medeiros, M. A. Musicante, and U. S. Costa, Journal of Computer Languages **68**, 101089 (2022).

[33] E. Verbitskaia, I. Kirillov, I. Nozkin, and S. Grigorev, in *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018 (Association for Computing Machinery, New York, NY, USA, 2018) p. 13вЂ“23.

[34] C. M. Medeiros, M. A. Musicante, and U. S. Costa, in *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*, SBLP '20 (Association for Computing Machinery, New York, NY, USA, 2020) p. 40вЂ“47.

[35] C. M. Medeiros, M. A. Musicante, and U. S. Costa, Journal of Computer Languages **51**, 75 (2019).

[36] P. G. Bradford and D. A. Thomas, RAIRO - Theoretical Informatics and Applications **43**, 567 (2009).

[37] P. G. Bradford, in *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)* (2017) pp. 247–253.

[38] C. B. Ward, N. M. Wiegand, and P. G. Bradford, in *2008 37th International Conference on Parallel Processing* (2008) pp. 373–380.

[39] Y. Susanina, in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20 (Association for Computing Machinery, New York, NY, USA, 2020) p. 2821вЂ“2823.

[40] E. Orachev, I. Epelbaum, R. Azimov, and S. Grigorev, in *Advances in Databases and Information Systems*, edited by J. Darmont, B. Novikov, and R. Wrembel (Springer International Publishing, Cham, 2020) pp. 49–59.

[41] Y. Lei, Y. Sui, S. H. Tan, and Q. Zhang, Proc. ACM Program. Lang. **7**, 10.1145/3591233 (2023).

[42] S. Chaudhuri, in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08 (Association for Computing Machinery, New York, NY, USA, 2008) p. 159вЂ“169.

[43] E. Scott and A. Johnstone, Science of Computer Programming **166**, 120 (2018).