# A Reachability Index for Recursive Label-Concatenated Graph Queries

Chao Zhang
Lyon 1 University
chao.zhang@univ-lyon1.fr

Angela Bonifati
Lyon 1 University
angela.bonifati@univ-lyon1.fr

Hugo Kapp
Oracle Labs
hugo.kapp@oracle.com

Vlad Ioan Haprian
Oracle Labs
vlad.haprian@oracle.com

Jean-Pierre Lozi
Oracle Labs
jean-pierre.lozi@oracle.com

## ABSTRACT

Reachability queries are fundamental operators for querying and processing graph data. They correspond to checking whether, given a source and a target node, these nodes are reachable in a given graph instance. Reachability queries in their simplest form without constraints have been extensively studied and can be efficiently evaluated in large-scale graphs. In this paper, we study the processing of recursive label-concatenated reachability (RLC) queries on an edge-labeled graph, where the reachability is defined as a regular expression using the Kleene plus on a concatenation of at most $k$ edge labels. TODO: Interesting in industry. Processing RLC queries require traversing cycles multiple times, depending on the specified label-constraint, which is not necessary for other types of reachability queries. We have found that current graph database engines cannot efficiently support RLC reachability queries, and little research has been conducted on them. We present the RLC index, which processes RLC queries by checking whether the source vertex can reach an intermediate vertex that can also reach the target vertex. We propose an indexing algorithm to build the RLC index, which guarantees soundness and completeness of query execution and avoids redundant index entries in the RLC index. TODO: Experimental results to be added.

**PVLDB Reference Format:**
Chao Zhang, Angela Bonifati, Hugo Kapp, Vlad Ioan Haprian, and Jean-Pierre Lozi. A Reachability Index for Recursive Label-Concatenated Graph Queries. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

## 1 INTRODUCTION

Graphs have been the natural choice of data representation in various domains [32], *e.g.,* social, biochemical, transportation networks, where the primary focus is to discover the relationships of entities. One of the fundamental operators to process graph data is the (plain) reachability query, *i.e.,* checking whether there exists a path from a source vertex $s$ to a target vertex $t$. Various indexing techniques have been proposed to efficiently process reachability queries over the simple form of a graph [5, 12, 13, 15, 16, 20, 22, 24–26, 36, 37, 39, 42, 43, 46].

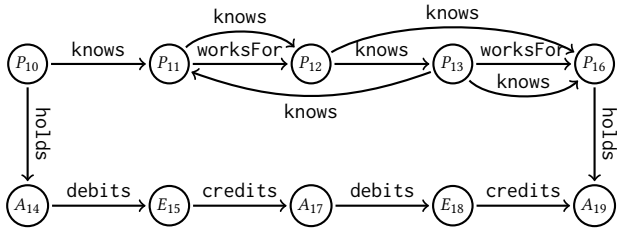


**Figure 1: An edge-labeled graph with nodes of persons, accounts, and external entities, and five edge labels: knows, worksFor, holds, debits, and credits.**

To facilitate the representation of different types of relationships in real-world applications, *edge-labeled graphs* and *property graphs* are more widely adopted than the simple form of a graph, where labels can be assigned to edges. In such advanced graph models, a labeled reachability query checks whether there is a path from $s$ to $t$ that can satisfy the path constraint specified by a regular expression over the edge labels along the path, which belongs to the category of *regular path queries* (RPQs) [7, 10]. Concatenation and the Kleene plus (or star) are two fundamental operators to form a regular expression. We refer to reachability queries with a path constraint of the Kleene plus over a concatenation of edge labels as *recursive label-concatenated queries* (RLC queries).

*Running example.* The graph in Fig. 1, inspired by a real business use case, shows an example of a social and professional network including the information of bank accounts of social peers. Navigating such graph by means of queries might bring interesting results, *e.g.,* identifying fraud and money laundering patterns among financial transactions. We leverage such a graph to instantiate concrete and real-life RLC queries. The query $Q1(A_{14}, A_{19}, (\texttt{debits, credits})^+)$ on the graph asks whether there is a path from $A_{14}$ to $A_{19}$ such that the label sequence of the path is a concatenation of an arbitrary number (one or more) of occurrences of (debits,credits). Queries like $Q1$ can be used to identify and investigate suspicious patterns of money transfers between account $A_{14}$ and $A_{19}$. The RLC query $Q1((A_{14}, A_{19}, (\texttt{debits,credits})^+)$ evaluates to *true* because of the existence of the path ($A_{14}$, debits, $E_{15}$, credits,


Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

$A_{17}$, debits, $E_{18}$, credits, $A_{19}$). Another example is the query $Q2(P_{10}, P_{13}, (\texttt{knows,worksFor})^+)$ that evaluates to false because there is no path from $P_{10}$ to $P_{13}$ satisfying the constraint $(\texttt{knows, worksFor})^+$.

RLC queries are frequently occurring in real-world query logs, *e.g.*, Wikidata Query Logs [11], which is the largest repository of open-source graph queries (of the order of $500M$ graph queries). In particular, RLC queries are often timeout in these logs [11] thus showing the limitations of graph query engines to efficiently evaluate these queries. Moreover, Neo4j(v4.3) [3] and TigerGraph(v3.3) [17], two of the main-stream graph data processing engines do not yet support RLC queries in their current version. However, these systems have already identified the need to support these queries in the near future by following the developments of the Standard Graph Query language (GQL) [2]. RLC queries can be expressed in Gremlin supported by TinkerPop-Enabled Graph Systems, in PGQL [41] supported by Oracle PGX [21, 38], and in SPARQL 1.1 (ASK query) supported by Virtuoso [4], Apache Jena [1], etc. However, many of these systems cannot efficiently evaluate RLC queries as also shown in our experimental study. In addition, to the best of our knowledge, little research has been carried out on the efficient evaluation of RLC queries.

In this work, we aim at studying the problem of efficiently processing RLC queries by designing an index that is suitable for these queries. Our RLC index leverages the 2-Hop labeling technique [16]. More precisely, in the RLC index, we assign to each vertex $v$ two sets $\mathcal{L}_{in}(v)$ and $\mathcal{L}_{out}(v)$, where $\mathcal{L}_{in}(v)$ contains vertices $u$ that can reach $v$ and (sub-)sequences of edge labels from $u$ to $v$, and $\mathcal{L}_{out}(v)$ contains vertices $w$ that are reachable from $v$ and (sub-)sequences of edge labels from $v$ to $w$. Then, the RLC index processes a RLC query with source $s$ and target $t$ by checking *(i)* whether there exists vertex $u$, such that $s$ can reach $u$ and $t$ is reachable from $u$, and *(ii)* whether the concatenation of the two recorded sequences, *i.e.*, ones from $s$ to $u$ and $u$ to $t$, can satisfy the label constraint of the query.

The major challenge for building an index to process RLC queries is that there can be infinite sequences of edge labels from vertex $s$ to vertex $t$ due to the presence of cycles on paths from $s$ to $t$. In this case, there will be infinite label sequences from $s$ to $t$ because the cycle can be traversed infinitely. Therefore, building an index for arbitrary RLC queries can be hard since the number of label sequences for cyclic graphs is exponential and potentially infinite. To overcome this issue, we introduce an input parameter $k$ in RLC queries, which is the number of concatenated labels, *e.g.*, $k = 2$ for $Q1(A_{14}, A_{19}, (\texttt{debits,credits})^+)$ as there are 2 labels concatenated. Interestingly, we found that $k$ usually has an upper-bound in practice. As an example, in massive real-world query logs [11], the length of concatenations with recursion is not larger than 3 and the total amount of queries with this characteristic is a large fraction of the property paths in the logs (roughly 65%). Moreover, many of these queries such as query $Q1$ above, are timeout in the logs, as they are complex to evaluate (NP-complete under the simple path semantics [8]). The RLC index is built with an input parameter $k$, and can answer any RLC query using a label concatenation of at most k labels. More precisely, we propose an indexing algorithm conducting backward and forward BFS from each vertex to build the RLC index. During each iteration of the BFS, we generate label constraints on the fly using $k$, which can be used to guide the subsequent search and avoid traversing cycles infinitely. In addition, we identify the situations where redundant traversals and index entries can be avoided and design corresponding rules for speeding up index construction and reducing index size.

*Contribution.* Our main contributions are summarized as follows:

- We introduce recursive label-concatenated reachability query (RLC query) that uses concatenation and the Kleene plus to form path constraints. We analyze the problem for building a reachability index for RLC queries, and propose a novel design based on an input parameter $k$ to overcome the underlying issue of cycle traversal.
- We propose the RLC index, the first reachability index for processing RLC queries, and a corresponding indexing algorithm. We formally proved that the constructed RLC index is sound and complete for a parameter $k$ given arbitrarily, where redundant index entries can be completely removed.
- TODO: Experimental results

We formally define RLC queries in Section 2. We present the theoretical foundation of the RLC index in Section 3, and the RLC index and the indexing algorithm with pruning rules in Section 4. Experimental results are presented in Section 5. We discuss related works in Section 6 and conclude in Section 7.

## 2 PROBLEM STATEMENT

An edge-labeled graph is $G = (V, E, \mathbb{L})$, where $V$ is a finite set of vertices, $E \subseteq V \times \mathbb{L} \times V$ a finite set of labeled edges, and $\mathbb{L}$ a finite set of labels. For the graph in Fig. 1, we have $\mathbb{L} = \{\texttt{knows, worksFor, debits, credits, holds}\}$, and $e_1$=($P_{10}$, knows, $P_{11}$) is a labeled edge. We use $\lambda : E \rightarrow \mathbb{L}$ to denote the mapping from an edge to its label, *e.g.*, $\lambda(e_1) = \texttt{knows}$. The frequently used symbols are summarized in Table 1.

### 2.1 Minimum Repeats

We use $l_i$ to denote an edge label, $L = (l_1, ..., l_n)$ a label sequence, $|L| = n$ the length of $L$, and $\epsilon$ the empty label sequence, *i.e.*, $|\epsilon| = 0$. The label sequence $L' = (l'_1, ..., l'_{n'})$ is a *repeat* of $L = (l_1, ..., l_n)$, if there exists an integer $z$, such that $\frac{n}{n'} = z \geq 1$, and $l'_j = l_{j+i \times n'}$ for every $j \in (1, ..., n')$ and $i \in (0, ..., z-1)$. A repeat $L'$ of $L$ is *minimum*, if $L'$ has the shortest length of all repeats of $L$. The *minimum repeat* (MR) of $L$ is denoted as $MR(L)$ that is also a sequence of edge labels. For example, given the path ($P_{10}$, knows, $P_{11}$, worksFor, $P_{12}$, knows, $P_{13}$, worksFor, $P_{16}$) in Fig. 1, we have $MR(p(P_{10}, P_{16})) = (\texttt{knows,worksFor})$. If $L = MR(L)$, we also say $L$ is a minimum repeat. Given a positive integer $k$ and a label sequence $L$, if $|MR(L)| \leq k$, then we say $L$ has a non-empty k-MR that is $MR(L)$. We use $\circ$ to denote the concatenation of label sequences (or labels), *i.e.*, $(l_1, ..., l_i) \circ (l_{i+1}, ..., l_n) = (l_1, ..., l_n)$, and $L \circ L = L^2$. For the empty label sequence $\epsilon$, we define $L \circ \epsilon = \epsilon \circ L = L$.

Lemma 2.1. *For a label sequence $L$, $MR(L)$ is unique.*

In this paper, we consider the *arbitrary paths* semantics [7], *i.e.*, allowing for duplicate vertices along the path, which can have an arbitrary length. An arbitrary path $p$ in $G$ is a vertex-edge alternating sequence $p(v_0, v_n) = (v_0, e_1, ..., e_n, v_n)$, where $n \geq 1$, and $v_i \in V, e_i \in E, i \in (0, ..., n)$, and $|p(v_0, v_n)| = n$ that is the length

of the path. For $p(v_0, v_n)$, $v_0$ is the source vertex and $v_n$ is the target vertex. If there exists a path from $v_0$ to $v_n$, then $v_0$ reaches $v_n$, denoted as $v_0 \rightsquigarrow v_n$. The label sequence of the path $p(v_0, v_n)$ is $\Lambda(p(v_0, v_n)) = (\lambda(e_1), ..., \lambda(e_n))$. When the context is clear, we also use $\Lambda(u, v)$ to denote the label sequence of a path from $u$ to $v$.

## 2.2 RLC Query

A label-constraint is $L^+ = (l_1, ..., l_k)^+$, where '+' is the Kleene plus, *i.e.*, one-or-more concatenations of the label sequence $L = (l_1, ..., l_k)$. W.l.o.g, we focus on a label-constraint $L$, s.t. $L = MR(L)$, *e.g.*, $L^+ =$ (knows,worksFor)$^+$. TODO: At the end of this paper, we discuss how to handle the case of $L \neq MR(L)$. A label sequence $\Lambda(u, v)$ of a path $p(u, v)$ satisfies a label-constraint $L^+$, if and only if $MR(\Lambda(u, v)) = L$. If such a path $p(u, v)$ exists, then we say $u$ can reach $v$ with the constraint $L^+$, denoted as $u \overset{L^+}{\rightsquigarrow} v$, otherwise $u \overset{L^+}{\not\rightsquigarrow} v$.

*Definition 2.2 (*RLC *reachability queries).* Given an edge-labeled directed graph $G = (V, E, \mathbb{L})$, a RLC query is a triple $(s, t, L^+)$, where $s, t \in V$, $L = MR(L)$, and $|L| \leq k$. If $s \overset{L^+}{\rightsquigarrow} t$, then the answer to the query is *true*. Otherwise, the answer is *false*.

Given a RLC query $Q(s, t, L^+)$, under the arbitrary path semantics, two naive approaches can be used to evaluate $Q$. The first approach is using an online traversal, *e.g.*, BFS, where each visited edge should satisfy the label (or state) transition of $L^+$, aka a finite automata-based approach. The second approach is pre-computing an extended transitive closure, where for each pair of vertices $(s, t)$ we record whether $s \rightsquigarrow t$, and all label sequences from $s$ to $t$. As demonstrated in our experiments, these two solutions require either too much query time or storage space, which are impractical for a large graph. Note that the naive approach to build a transitive closure is not feasible in our case because of cycles on the path from $s$ to $t$. We adopt a variant of transitive closure equipped with kernels (presented in Section 3). We describe in detail this variant of transitive closure in Section 5.

## 2.3 Indexing Problem

Our goal is to build an index to efficiently process RLC queries. The corresponding indexing problem is summarized as follows.

Problem 2.1. *Given an edge-labeled graph $G$, the indexing problem is to build a reachability index for processing* RLC *queries on $G$, such that the storage of label sequences in the index is minimal and the correctness of query processing is preserved.*

We firstly observe that recording MRs, instead of raw label sequences of paths in $G$, can reduce the storage space, and such a strategy does not violate the correctness of query processing. The main benefits are twofold: (1) MRs are not longer than raw label sequences; (2) different raw label sequences may have the same MR. For example, in Fig. 1, there exist two paths from $P_{10}$ to $P_{16}$ having the label sequence (knows,knows,knows,knows) and (knows,knows,knows), which have the same MR, *i.e.*, knows.

*Definition 2.3 (Concise Label Sequences).* Let $\mathbb{P}(s, t)$ be the set of all paths from $s$ to $t$. The concise set of label sequences from vertex $s$ to $t$, denoted as $S^k(s, t)$, is the set of k-MRs of all label sequences from $s$ to $t$, *i.e.*, $S^k(s, t) = \{L | p \in \mathbb{P}(s, t), MR(\Lambda(p)) = L, |L| \leq k\}$.

**Table 1: Frequently used symbols.**

| Notation | Description |
|---|---|
| $p$, or $p(u, v)$ | a path, or the path from $u$ to $v$ |
| $\circ$ | concatenation of labels or label sequences |
| $\Lambda(u, v)$, or $\Lambda(p(u, v))$ | the label sequence of a path from $u$ to $v$ |
| $L$ | a label sequence |
| $L^+$ | a label constraint |
| $MR(L)$ | the minimum repeat of a label sequence $L$ |
| $k$ | the upper bound of the number of labels in a $L^+$ |
| $S^k(u, v)$ | the concise set of minimum repeats from $u$ to $v$ |
| $u \overset{L^+}{\rightsquigarrow} v$, or $u \overset{L^+}{\not\rightsquigarrow} v$ | $u$ reaches $v$ through an $L^+$-path, or otherwise |
| $u \rightsquigarrow v$, or $u \not\rightsquigarrow v$ | $u$ reaches $v$, or otherwise |
| $in(v)$, or $out(v)$ | the set of vertices that can reach $v$, or $v$ can reach |
| $aid(v)$ | the access id of vertex $v$ by the indexing algorithm |

To deal with RLC queries, we need to compute and record the concise label sequences.

Proposition 2.4. *$s \overset{L^+}{\rightsquigarrow} t, |L| \leq k$ in $G$ if and only if $L \in S^k(s, t)$.*

For example, in Fig. 1, we have $S^2(P_{12}, P_{16}) = \{$(knows), (knows, worksFor)$\}$. With $S^2(P_{12}, P_{16})$, RLC queries with $P_{12}$ as source and $P_{16}$ as target can be processed correctly.

# 3 KERNEL-BASED SEARCH

In this section, we deal with the following question: *how to compute concise label sequences?* The problem for computing a concise label sequence is that if a cycle exists on a path from $s$ to $t$, there exist infinite paths from $s$ to $t$, which makes the computation of $S^k(s, t)$ infeasible, *e.g.*, $|\mathbb{P}(P_{11}, P_{13})|$ in Fig. 1 is infinite. We overcome this issue by leveraging the upper bound of concatenated labels in a constraint, *i.e.*, $k$. We observed that we don't have to compute all possible label sequences for paths going from $P_{11}$ to $P_{13}$, as the set of label sequences $L$ such that $|MR(L)| <= k$ is actually finite. Specifically, let $v$ be an intermediate vertex that a forward breadth-first search from $s$ is visiting. The main idea is that when the path from $s$ to $v$ reaches a specific length, we can decide whether we need to further explore the outgoing neighbours of $v$. Moreover, if the outgoing neighbours of $v$ are worth exploring, the following search can be guided by a specific label constraint. In the following, we first provide an illustrating example, and then formally define the specific constraint that is used to guide search.

*Example 3.1 (Illustrating Example).* Consider the graph in Fig. 1. Assuming we need to compute $S^2(P_{11}, P_{13})$, *i.e.*, $k = 2$. When $P_{13}$ is visited for the first time, we add (knows) and (worksFor, knows) into $S^2(P_{11}, P_{13})$. After that, when the depth of search reaches $2k = 4$, *i.e.*, $P_{12}$ is visited for the second time, we have 4 different label sequences, which are (knows, knows, knows, knows), (knows, knows, knows, worksFor), (worksFor, knows, knows, knows), and (worksFor, knows, knows, worksFor). Given this, all the 4 label sequences except the first one does not need to be expanded anymore, because their expansions can not have a non-empty k-MR, *i.e.*, a MR whose length is not larger than 2. Then the following search is guided by (knows)$^+$ that is computed from (knows, knows,

knows, knows). However, because there already exists (knows) in $S^2(P_{11}, P_{13})$, the search does not need to continue.

*Definition 3.2 (Kernel and Tail).* If a label sequence $L$ can be represented as $L = (L')^h \circ L''$, where $h \geq 2$, and $L'$ and $L''$ are two label sequences, such that $L' \neq \epsilon$ and $MR(L') = L'$, and $L''$ is $\epsilon$ or a proper prefix of $L'$, then $L$ has the *kernel* $L'$ and the *tail* $L''$.

For example, the label sequence (knows, knows, knows, knows) from $P_{11}$ has a kernel knows and a tail $\epsilon$.

*Kernel-based search.* When a kernel has been determined at a vertex that is being visited, the subsequent search to compute $S^k(s,t)$ can be guided by the Kleene plus of the kernel, *e.g.*, since $P_{12}$ is visited, $(\text{knows})^+$ is used to guide the search in Example 3.1. We call this strategy KBS (*kernel-based search*) in the remainder of this paper. In a nutshell, KBS consists of two phases: (1) *kernel-search* and (2) *kernel-BFS*, where the first phase is to compute kernels, and the second to perform kernel-guided BFS. We show in Theorem 3.4 that KBS can compute a sound and complete $S^k(s,t)$. Before that, we fist show the kernel of a label sequence is unique in Lemma 3.3 that will be used in the proof of Theorem 3.4.

Lemma 3.3. *If $L$ has a kernel, then the kernel is unique.*

Proof. The proof is based on induction. The statement is if a label sequence $L$ of length $n$ has a kernel, then the kernel is unique. It is trivial to prove the initial case $|L| = 2$. Assuming the case $n$ is true, then we show the case $n+1$ is also true. Let $|L| = n+1$. We use $\bar{L}$ to denote the label sequence obtained by removing the last label of $L$, *i.e.*, $|\bar{L}| = n$. We proof the case $n+1$ below.

Assuming $L$ can have two different kernels $L_1$ and $L_2$, and $L_1 \neq L_2$, then $L = (L_1)^{h_1} \circ L'_1, h_1 \geq 2$ and $L = (L_2)^{h_2} \circ L'_2, h_2 \geq 2$. If $|L'_1| = 0$ and $|L'_2| = 0$, then $L$ has two MRs, which is contradictory (Lemma 2.1). If $|L'_1| \neq 0$ and $|L'_2| \neq 0$, then $\bar{L}$ has kernels $L_1$ and $L_2$, which contradicts to the case $n$. The remaining cases are that only one of $L'_1$ and $L'_2$ has a length of 0. W.l.o.g, consider $|L'_1| = 0$ and $|L'_2| \neq 0$. Given this, if $h_1 > 2$, then $\bar{L}$ still has kernels $L_1$ and $L_2$, which is contradictory. Then we have $h_1 = 2$ and $|L'_1| = 0$, *i.e.*,

$$L = L_1 \circ L_1. \tag{1}$$

In addition, we have

$$L = (L_2)^{h_2} \circ L'_2, h_2 \geq 2, |L'_2| \neq 0. \tag{2}$$

Let $L = (l_1, ..., l_{2|L_1|})$ and $|L_1| = a|L_2| + b, 1 \leq a, b < |L_2|$. According to Equ. (1), we have $l_i = l_{i+|L_1|}, 1 \leq i \leq |L_1|$ and $l_{i'} = l_{i'-|L_1|}, |L_1| < i' \leq 2|L_1|$, which means $l_i = l_{i+a|L_2|+b}$ and $l_{i'} = l_{i'-a|L_2|-b}$. Based on Equ. (2), we have $l_i = l_{i+b}$ and $l_{i'} = l_{i'} - b$. Given this, consider the following two cases: case (i) if $2|L_1| \bmod b = 0$, then $|MR(L_1 \circ L_1)| = b \neq |L_1|$ that contradicts to the fact that $L_1$ is the unique MR of $L$; case (ii) if $2|L_1| \bmod b \neq 0$, then $L = (L_3)^{h_3} \circ L'_3$, where either $|L_3| = b, h_3 \geq 2$, and $|L'_3| \neq 0$, or $|L_3| < b$ and $h_3 > 2$. Note that, in the two sub-cases of case (ii), $L'_3$ is $\epsilon$, or a proper prefix of $L_3$. Therefore, we have $\bar{L}$ has a kernel $L_3$, $|L_3| \leq b$. However, $\bar{L}$ also has a kernel $L_2$ and $|L_2| > b \geq |L_3|$, which is also a contradiction. □

Theorem 3.4. *Given a path $p$ from $u$ to $v$ and a positive integer $k$. Then $p$ has a non-empty k-MR if and only if one of the following conditions is satisfied,*

- *Case 1: $|p| \leq k$. $MR(\Lambda(p))$ is the k-MR of $p$;*
- *Case 2: $k < |p| \leq 2k$. If $|MR(\Lambda(p))| \leq k$, $MR(\Lambda(p))$ is the k-MR of $p$;*
- *Case 3: $|p| > 2k$. Let $x$ be the intermediate vertex on $p$, s.t. $|p(u,x)| = 2k$. If $\Lambda(p(u,x))$ has a kernel $L'$ and a tail $L''$, and $MR(L'' \circ \Lambda(p(x,v))) = L'$, then $L'$ is the k-MR of $p$.*

Proof. It is not difficult to prove Case 1 and Case 2. We focus on Case 3 below. For ease of presentation, let $\Lambda(u,x) = \Lambda(p(u,x))$ and $\Lambda(x,v) = \Lambda(p(x,v))$.

(Sufficiency) Because $\Lambda(u,x)$ has the kernel $L'$ and the tail $L''$, such that $\Lambda(u,x) = (L')^h \circ L'', h \geq 2$. Thus, we have $MR(\Lambda(u,x) \circ \Lambda(x,v)) = MR((L')^h \circ L'' \circ \Lambda(x,v)) = L'$, otherwise $MR(L'' \circ \Lambda(x,v)) \neq L'$. In addition, we have $|L'| \leq k$ because $|\Lambda(u,x)| = 2k$. Thus, $L'$ is the k-MR of $p$.

(Necessity) We show that $p$ does not have a non-empty k-MR in the following two cases.

- Case (i): $\Lambda(u,x)$ does not have a kernel and a tail. Assuming $p$ can have a non-empty k-MR $L'''$ in this case. Because $|L'''| \leq k$ and $|\Lambda(u,x)| = 2k$, such that $\Lambda(u,x)$ has a kernel and a tail, which contradicts to the case definition.
- Case (ii): $\Lambda(u,x)$ has a kernel $L'$ and a tail $L''$, but $MR(L'' \circ \Lambda(x,v)) \neq L'$. Assuming $p$ has a non-empty k-MR $L'''$ in this case. Knowing that $|L'''| \leq k$ and $|\Lambda(u,x)| = 2k$, such that $L''' = L'$ because the kernel of $\Lambda(u,x)$ is unique (Lemma 3.3). Therefore, $MR((L')^h \circ L'' \circ \Lambda(x,v)) = L', h \geq 2$, which means $MR(L'' \circ \Lambda(x,v)) = L'$ that is also a contradiction.

□

In Case 3 of Theorem 3.4, if $\Lambda(u,x)$ of $p$ does not have a kernel, then $p$ does not have a non-empty k-MR. Otherwise the k-MR of $p$ can only be the kernel of $\Lambda(u,x)$. In another sense, Theorem 3.4 says that although $|p(u,v)|$ can be very large or infinite, we can determine the possible k-MRs of $p$ by applying a search up to $2k$ length from $u$. In addition, with kernels determined in Theorem 3.4, we will not miss any k-MRs for paths from $u$ to $v$.

We discuss below two strategies to compute kernels based on Theorem 3.4, namely *lazy* KBS and *eager* KBS, and explain why eager KBS is better than lazy KBS, which is used in our indexing algorithm presented later in Section 4.3.

*Lazy KBS.* Theorem 3.4 can be transformed into an algorithm to find kernels, *i.e.*, for a source vertex we generate all paths of length $2k$, and then compute all the kernels of these paths. This strategy is referred to as lazy KBS, which means kernels are correctly determined when the length of paths reaches $2k$, *e.g.*, lazy KBS is used in Example 3.1.

*Eager KBS.* In contrast to the lazy strategy, we can determine kernel candidates earlier, instead of valid kernels that requires the length of paths to be $2k$. The main idea is to treat any k-MR that is computed using any path $p, |p| \leq k$ as a kernel candidate, and then kernel candidates will be used to guide subsequent search. As KBS is a breadth-first search, such that the set of kernel candidates does not miss any valid kernels. Although a false kernel may be included, the search guided by the false kernel will not reach a target vertex through a path of which the k-MR is the false kernel. Therefore, the set of concise label sequences computed by the eager strategy is still sound and complete.

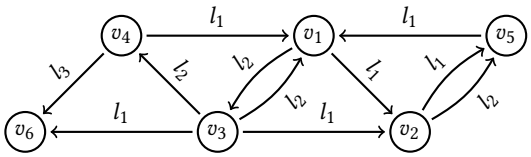


**Figure 2: Running example.**

*Example 3.5.* Consider the example of computing $S^2(P_{10}, P_{13})$ in Fig. 1. With eager strategy, two kernel candidates can be determined when $P_{12}$ is visited for the first time, namely (knows) and (knows, worksFor). Although a false kernel (knows, worksFor) is included, the search guided by (knows, worksFor)$^+$ cannot reach $P_{13}$ through a path that has the k-MR (knows, worksFor).

The key advantage of the eager strategy over the lazy strategy is that it allows us to advance KBS from the kernel-search phase to the kernel-BFS phase. This can make KBS more efficient because generating all paths of length $2k$ from a source vertex is more expensive than generating only paths of length $k$, especially on dense graphs. In addition, vertices that have been visited can be marked in the kernel-BFS phase, avoiding unnecessary traversals.

## 4 RLC INDEX

In this section, we present the RLC index, and also the corresponding query and indexing algorithm.

### 4.1 Overarching Idea

Given a RLC query $(s, t, L^+), |L| \leq k$, the idea is to check whether there exists a 2-hop path $(s, ..., u, ..., t)$ whose label sequence satisfies the label constraint $L^+$, where $u$ is an intermediate vertex in $p$. In another sense, the query is answered by concatenating two MRs of sub-paths of $p$, *i.e.*, $MR(\Lambda(s, u))$ and $MR(\Lambda(u, t))$.

*Definition 4.1 (*RLC *Index).* Let $G = (V, E, \mathbb{L})$ be an edge-labeled graph and $k$ be a positive integer. The RLC index of $G$ assigns to each vertex $v \in V$ two sets: $\mathcal{L}_{in}(v) = \{(u, L') | u \rightsquigarrow v, L' \in S^k(u, v)\}$, and $\mathcal{L}_{out}(v) = \{(w, L'') | v \rightsquigarrow w, L'' \in S^k(v, w)\}$. Therefore, there is a path $p(s, t)$ satisfying an arbitrary constraint $L^+, |L| \leq k$, if and only if one of the following cases is satisfied,

- Case 1: $\exists (x, L') \in \mathcal{L}_{out}(s)$ and $\exists (x, L'')) \in \mathcal{L}_{in}(t)$, such that $L' = L'' = L$;
- Case 2: $\exists (t, L''') \in \mathcal{L}_{out}(s)$, or $\exists (s, L''') \in \mathcal{L}_{in}(t)$, such that $L''' = L$.

The size of the RLC index is defined to be $\sum_{v \in V} |\mathcal{L}_{out}(v)| + |\mathcal{L}_{in}(v)|$.

*Example 4.2 (Running Example of* RLC *Index).* Consider the graph $G$ shown in Fig. 2. The RLC index with $k = 2$ for $G$ is presented in Table 2. We have $Q_1(v_3, v_6, (l_2, l_1)^+) = true$ because $\exists (v_1, (l_2, l_1)) \in \mathcal{L}_{out}(v_3)$ and $\exists (v_1, (l_2, l_1)) \in \mathcal{L}_{in}(v_6)$. Indeed, there exists the path $(v_3, l_2, v_4, l_1, v_1, l_2, v_3, l_1, v_6)$ from $v_3$ to $v_6$ in the graph in Fig. 2. For $Q_2(v_1, v_2, (l_2, l_1)^+)$, the answer is *true* because $\exists (v_1, (l_2, l_1)) \in \mathcal{L}_{in}(v_2)$. Given $Q_3(v_1, v_3, (l_1)^+)$, we have the answer is $false$. Although $v_1$ can reach $v_3$, *e.g.*, $\exists (v_1, l_2) \in \mathcal{L}_{in}(v_3)$, the constraint $(l_1)^+$ of $Q_3$ cannot be satisfied.

Given $G = (V, E, \mathbb{L})$, the minimum RLC index is the one with the minimum $\sum_{v \in V} |\mathcal{L}_{out}(v)| + |\mathcal{L}_{in}(v)|$. Finding the *minimum* RLC index is NP-hard, because it can become a 2-Hop labeling when $\mathbb{L}$ contains only one label, and finding the minimum 2-Hop labeling is NP-hard [16]. Although it is expensive to find the minimum RLC index, it is still worthwhile to remove as many redundant index entries as possible. The intuition is that if there exists a path $p$ such that $u \overset{L^+}{\rightsquigarrow} v$, then the RLC index only records the reachability information of path $p$ once, *i.e.*, either through Case 1 or Case 2 in Definition 4.1.

**Table 2: The RLC index for the graph in Fig. 2.**

| V | $\mathcal{L}_{in}(v)$ | $\mathcal{L}_{out}(v)$ |
|---|---|---|
| $v_1$ | - | $(v_1, l_2)$, $(v_1, l_1)$, $(v_1, (l_2, l_1))$ |
| $v_2$ | $(v_1, l_1)$, $(v_1, (l_2, l_1))$ | $(v_1, (l_2, l_1))$, $(v_1, l_1)$ |
| $v_3$ | $(v_1, l_2)$, $(v_1, (l_1, l_2))$ | $(v_1, l_2)$, $(v_1, (l_2, l_1))$, $(v_1, l_1)$, $(v_3, (l_1, l_2))$ |
| $v_4$ | $(v_1, l_2)$ | $(v_1, l_1)$, $(v_3, (l_1, l_2))$ |
| $v_5$ | $(v_1, (l_1, l_2))$, $(v_1, l_1)$,$(v_3, (l_1, l_2))$, $(v_2, l_2)$ | $(v_1, l_1)$, $(v_3, (l_1, l_2))$ |
| $v_6$ | $(v_1, (l_2, l_1))$, $(v_3, l_1)$, $(v_3, (l_2, l_3))$, $(v_4, l_3)$ | - |

*Definition 4.3 (Condensed* RLC *Index).* The RLC index is condensed, if for every index entry $(s, L) \in \mathcal{L}_{in}(t)$ or $(t, L) \in \mathcal{L}_{out}(s)$, there do not exist index entry $(u, L') \in \mathcal{L}_{out}(s)$ and $(u, L'') \in \mathcal{L}_{in}(t)$ such that $L = L' = L''$.

We focus on designing an indexing algorithm that can build a correct (sound and complete) and condensed RLC index.

### 4.2 Query Algorithm

The query algorithm is presented in Algorithm 1, where we use $I$ to denote an index entry. Each index entry $I$ has the schema $(vid, mr)$, where $vid$ represents vertex id and $mr$ recorded minimal repeat. Given a RLC query $(s, t, L^+)$, to efficiently find $(u, L') \in \mathcal{L}_{out}(s)$ and $(u, L'') \in \mathcal{L}_{in}(t)$, we execute a merge join over $\mathcal{L}_{out}(s)$ and $\mathcal{L}_{in}(t)$, shown at line 4 in 1. The output of the merge join is a set of index entry pairs $(I', I'')$, such that $I'.vid = I''.vid$. Index entries are sorted in ascending order in advance by $vid$. Case 1 of the RLC index (see Definition 4.1) is checked at line 1, and Case 2 is checked at line 2. If one of these cases can be satisfied, the answer true will be returned immediately. Otherwise, index entries in $\mathcal{L}_{out}(s)$ and $\mathcal{L}_{in}(t)$ are exhaustively merged, and the answer false will be returned at last.

### 4.3 Indexing Algorithm

In this subsection, we present an indexing algorithm (Algorithm 2) to build the RLC index which is sound, complete, and condensed. We use $v_i$ to denote a vertex with id $i$. Given a graph $G(V, E, \mathbb{L})$, the indexing algorithm mainly performs backward and forward KBS from each vertex in $V$ to create index entries, and pruning rules are applied to accelerate index building.

*4.3.1 Indexing with KBS.* We explain below how the backward KBS creates $\mathcal{L}_{out}$-entries. The forward KBS follows the same procedure, except that $\mathcal{L}_{in}$-entries will be created. The backward KBS from vertex $v_i$ tries to create $\mathcal{L}_{out}$-entries. Suppose KBS is visiting $v$. If $|MR(\Lambda(v, v_i))| \leq k$, then we add $(v_i, MR(\Lambda(v, v_i)))$ into

**Algorithm 1:** Query Algorithm

```
1 procedure Query(s, t, L+)
2     if ∃(t, L) ∈ L_out(s) or ∃(s, L) ∈ L_in(t) then
3         return true;
4     for (I', I'') ∈ mergeJoin(L_out(s), L_in(t)) do
5         if I'.mr = L and I''.mr = L then
6             return true;
7     return false;
```

$\mathcal{L}_{out}(v)$. Although there may be cycles in a graph, KBS will not go on forever, because when the depth of search reaches $k$, KBS will be transformed into the kernel-BFS phase that is guided by the Kleene plus of kernel candidates, such that KBS terminates if any invalid label (or state) transition is met, or a vertex has been visited with the same label (or state).

KBSs are executed from each vertex in $V$ and this execution follows a specific order. The idea is to start with vertices that have more connections to other vertices. In the RLC index, we use the IN-OUT strategy, *i.e.*, sorting vertices according to $(|out(v)|+1) \times (|in(v)|+1)$ in descending order, which has shown to be an efficient and effective strategy in the literature of 2-hop labeling framework [6, 35, 45]. The id of vertex $v$ in the sorted list is referred to as access id, denoted as $aid(v)$ starting from 1, *e.g.*, for the graph in Fig. 2, the sorted list is $(v_1, v_3, v_2, v_4, v_5, v_6)$, where $aid(v_1) = 1$ and $aid(v_3) = 2$.

*Example 4.4 (Running Example of Indexing).* Consider the graph in Fig. 2 and the RLC index in Table 2 with $k = 2$. The KBSs are executed from each vertex in the order of $(v_1, v_3, v_2, v_4, v_5, v_6)$. We explain the backward KBS from $v_1$ as follows. The traversal of depth 1 of this backward KBS visits $v_4$ and creates $(v_1, l_1)$ in $\mathcal{L}_{out}(v_4)$, visits $v_3$ and creates $(v_1, l_2)$ in $\mathcal{L}_{out}(v_3)$, and visits $v_5$ and creates $(v_1, l_1) \in \mathcal{L}_{out}(v_5)$. The traversal of depth 2 creates $(v_1, (l_2, l_1))$ in $\mathcal{L}_{out}(v_3)$, $(v_1, l_2)$ in $\mathcal{L}_{out}(v_1)$, $(v_1, l_1) \in \mathcal{L}_{out}(v_2)$, and $(v_1, (l_2, l_1)) \in \mathcal{L}_{out}(v_2)$. Then the kernel-search phase of this KBS terminates because the depth of the search reaches 2, which generates kernel candidate $l_1$ with a set of frontier vertices $\{v_4, v_5, v_2\}$, kernel candidate $l_2$ with a set of frontier vertices $\{v_3, v_1\}$, and kernel candidate $(l_2, l_1)$ with a set of frontier vertices $(v_3, v_2)$. After this, this KBS is turned into three kernel-BFSs guided by $(l_1)^+$, $(l_2)^+$, and $(l_2, l_1)^+$ with the corresponding frontier vertices. The kernel-BFS terminates under the case of an invalid label transition or a repeated visiting. For example, the label of the incoming edge of $v_3$ is $l_2$, which is an invalid state transition of $(l_2, l_1)^+$ in a backward KBS from $v_1$, such that the kernel-BFS guided by $(l_2, l_1)^+$ terminates at $v_3$. For another example, index entry $(v_1, l_1) \in \mathcal{L}_{out}(v_1)$ is created when $v_1$ is visited for the first time by the kernel-BFS guided by $(l_1)^+$, but this kernel-BFS will not continue when it visits $v_5$ that has already been visited.

*4.3.2 Pruning Rules.* To accelerate index construction as well as remove redundant index entries, we apply pruning rules during KBS. For ease of presentation, we present the pruning rules for backward KBSs, and the same is true for forward ones.

- ***PR1:*** *If the k-MR of an index entry that needs to be recorded can be acquired from the current snapshot of the* RLC *index, then the index entry can be skipped.*
- ***PR2:*** *If vertex $v_i$ is visited by the backward KBS performed from vertex $v_{i'}$ s.t. $aid(v_{i'}) > aid(v_i)$, then the corresponding index entry can be skipped.*
- ***PR3:*** *During the backward kernel-BFS, if vertex $v_i$ is visited by the backward kernel-BFS performed from vertex $v_{i'}$, and PR1 or PR2 is triggered, then vertex $v_i$ and $in(v_i)$ can be skipped.*

Note that if PR2 is triggered then PR1 must be triggered, because a path $p$ can be visited by either the forward KBS from the source of $p$ or the backward KBS from the target of $p$. However, checking for PR2 only requires the access id of vertices instead of evaluating a query using a snapshot of the RLC index. This is why we extract PR2 from PR1. The correctness of the indexing algorithm with pruning rules is guaranteed by Theorem 4.10 presented in Section 4.4.

*Example 4.5 (Running Example of Pruning Rules).* Consider the forward KBS from $v_3$ for the graph in Fig. 2. It can visit $v_2$ through label sequence $(l_2, l_1)$, such that it tries to create $(v_3, (l_2, l_1))$ in $\mathcal{L}_{in}(v_2)$. However, there already exist $(v_1, (l_2, l_1)) \in \mathcal{L}_{out}(v_3)$ and $(v_1, (l_2, l_1)) \in \mathcal{L}_{in}(v_2)$, such that $Q(v_3, v_2, (l_2, l_1)^+) = true$ with the current snapshot of the RLC index, *i.e.*, the reachability information has already been recorded. Therefore, the index entry $(v_3, (l_2, l_1))$ in $\mathcal{L}_{in}(v_2)$ is pruned according to PR1. As an example of PR2, consider the backward KBS from $v_2$. It can visit $v_1$ through path $(v_1, l_2, v_3, l_1, v_2)$, such that it tries to create $(v_2, (l_2, l_1))$ in $\mathcal{L}_{out}(v_1)$. Given $aid(v_2) > aid(v_1)$, such that the index entry can be pruned by PR2. As an example of PR3, consider the forward KBS from $v_2$. It visits $v_2$ through path $(v_2, l_2, v_5, l_1, v_1, l_2, v_3, l_1, v_2)$, where at $v_5$ the KBS is transformed from a kernel-search to a kernel-BFS guided by $(l_2, l_1)^+$. When $v_2$ visits itself for the first time, the KBS tries to create index entry $(v_2, (l_2, l_1)) \in \mathcal{L}_{in}(v_2)$, which can be pruned by PR1 because of $(v_1, (l_2, l_1)) \in \mathcal{L}_{out}(v_2)$ and $(v_1, (l_2, l_1)) \in\in (v_2)$. In this case, PR3 is triggered also, which means $v_2$ and $out(v_2)$ is skipped by this kernel-BFS.

The indexing algorithm is presented in Algorithm 2. For ease of presentation, each procedure focuses on the backward case, and the forward case can be obtained by trivial modifications, *e.g.*, replacing in-coming edges with out-going edges. We use the KMP algorithm [27] to compute the minimum repeat of a label sequence, *i.e.* MR() at line 13 in Algorithm 2. The indexing algorithm performs backward and forward KBS from each vertex. The KBS from a vertex $v$ consists of two phases: kernel-search (line 6 to line 17) and kernel-BFS (line 24 to line 37). The kernel-search returns for each vertex $v$ all kernel candidates and a set of frontier vertices $vSet$. The kernel-BFS is performed for each kernel candidate, *i.e.*, a BFS with vertices in $vSet$ as frontier vertices guided by a kernel candidate. PR1 and PR2 are included at line 19, which can be triggered by both kernel-search and kernel-BFS. PR3 implemented at line 31, on the other hand, can only be triggered by kernel-BFS.

*Remark.* An alternative version of the RLC index allowed to concatenate different minimum repeats to answer a RLC query, *i.e.*, in Case 1 of Definition 4.1, $L'$ can be different from $L''$ in the initial version. However, such a design will prevent the use of PR3, which can prune vertices and avoid redundant traversals. Consequently, the indexing time of the alternative version is much longer than

**Algorithm 2:** Indexing Algorithm.

```
1  procedure kernelBasedSearch(v, k)
2      for (L, vSet) ∈ backwardKernelSearch(v, k) do
3          backwardKernelBFS(v, vSet, L);
4      for (L, vSet) ∈ forwardKernelSearch(v, k) do
5          forwardKernelBFS(v, vSet, L);
6  procedure backwardKernelSearch(v, k)
7      q ← an empty queue of (vertex, label sequence);
8      q.enqueue(v, ϵ);
9      map ← a map of (kernel candidates, vertex set);
10     while q is not empty do
11         (x, seq) ← q.dequeue();
12         for in-coming edge e(y, x) to x do
13             seq′ ← λ(e(y, x)) ∘ seq; L ← MR(seq′);
14             insert (y, v, L); map.get(L).add(x);
15             if |seq′| < k then
16                 q.enqueue(y, seq′);
17     return map;
18 procedure insert(s, t, L)
19     if aid(t) > aid(s) or Query(s, t, L+) then // PR 2 or 1
20         return false;
21     else
22         add (t, L) into L_out(s);
23         return true;
24 procedure backwardKernelBFS(v, vSet, L)
25     q ← an empty queue of (vertex, integer);
26     for x ∈ vSet do
27         mark x as visited by state 1, q.enqueue(x, |L|);
28     while q is not empty do
29         (x, i) ← q.dequeue(), i ← i − 1;
30         if i = 0 then i = |L| ;
31         label l ← L.get(i);
32         for in-coming edge e(y, x) to x do
33             if l ≠ λ(e(y, x)) or y was visited by state i then
34                 continue;
35             if i = 1 and insert(y, v, L) then      // PR 3
36                 continue;
37             q.enqueue(y, i); mark y visited by state i;
```

the version introduced in this paper, *e.g.*, even for the smallest graph used in our experiment (AD graph presented in Section 5), the indexing time of the alternative indexing is 32X slower than the current one. Therefore, we focus on concatenating the same minimum repeats, as shown in Case 1 of Definition 4.1.

## 4.4 Correct and Condensed RLC Index

We present in Theorem 4.9 that pruning rules can guarantee the condensed property of the RLC index, and in Theorem 4.10 that the RLC index constructed by Algorithm 2 is correct, *i.e.*, sound and complete. Before proceeding further, we first present the following lemmas that will be used to proof the two theorems.

Lemma 4.6. *Given a path $p(s,t)$ having a k-MR $L$. If the KBS from $s$ can visit $t$ (or the KBS from $t$ can visit $s$), then the k-MR $L$ of $p(s,t)$ must be recorded in the index.*

Proof. If the KBS from $s$ can visit $t$, then regardless of whether PR1 or PR2 is applied, the k-MR $L$ of $p(s,t)$ must be recorded. □

Lemma 4.7. *Given two paths $p(s,u)$ and $p(u,t)$ with k-MR $L$ in a graph, where $aid(u) < aid(s)$ and $aid(u) < aid(t)$. We have: if $aid(u) \leq i$, then the k-MR of the path from $s$ to $t$ through vertex $u$ is recorded by Algorithm 2 in the $i$-th snapshot of the* RLC *index that is computed after performing KBS from a vertex with access id $i$.*

Proof. The proof is based on induction. It is trivial to prove the initial case $i = 1$. We assume the case $i = n$ is true and prove the case $i = n + 1$ below. We only need to show the case $aid(u) = n + 1$. Let $p(s,t) = (s, ..., u, ..., t)$.

Assuming the backward KBS from $u$ does not visit $s$. Then PR3 is triggered, such that there exists vertex $w$, $aid(w) < aid(u)$, and $p(s,w)$ and $p(w,u)$ have the k-MR $L$. This case can be reduced to the case $i = n$, because the k-MR of path $(w, ..., u, ..., t)$ is $L$ and $aid(w) < aid(u) < n + 1$. In the same way, we can also prove the case if the forward KBS from $u$ does not visit $t$.

We consider the case that both the backward KBS and the forward KBS from $u$ can visit $s$ and $t$. For $p(s,u)$, we have the following two cases: Case (1) $\exists(u,L) \in \mathcal{L}_{out}(s)$; Case (2) $\exists(v,L) \in \mathcal{L}_{out}(s)$ and $\exists(v,L) \in \mathcal{L}_{in}(u)$, $aid(v) < aid(u)$. For Case (2), both $p(s,v)$ and $p(v,t)$ have k-MR $L$, such that this case can be reduced to the case $i = n$ as $aid(v) < aid(u) = n + 1$. Then we only need to consider Case (1), *i.e.*, $\exists(u,L) \in \mathcal{L}_{out}(s)$. In the same way, for $p(u,t)$, we only need to consider the case $\exists(u,L) \in \mathcal{L}_{in}(s)$. Given this, we have the k-MR of the path from $s$ to $t$ is recorded by having $(u,L) \in \mathcal{L}_{out}(s)$ and $(u,L) \in \mathcal{L}_{in}(t)$. □

Lemma 4.8. *Given a path $p$ from $s$ to $t$ with a k-MR $L$. If the index entry $(t,L) \in \mathcal{L}_{out}(s)$ (or $(s,L) \in \mathcal{L}_{in}(t)$) is pruned because of PR3, then we have one of the following two cases:*

- *$\exists(s,L) \in \mathcal{L}_{in}(t)$ (or $\exists(t,L) \in \mathcal{L}_{out}(s)$);*
- *$\exists(v,L) \in \mathcal{L}_{out}(s)$ and $\exists(v,L) \in \mathcal{L}_{in}(t)$, such that $aid(v) < aid(t)$ (or $aid(v) < aid(s)$).*

Proof. We proof the case of $aid(t) \leq aid(s)$. The proof for the other case follows the same sketch. Let $p(s,t) = (s, ..., u, ..., t)$, such that PR3 can be triggered. W.l.o.g, let $aid(u) < aid(t)$ (if $aid(u) \geq aid(t)$ and PR3 is triggered, then there exists vertex $w, aid(w) < aid(u)$, which can be reduced to the case of $aid(u) < aid(t)$). Given this, we have path $p(s,u)$ and $p(u,t)$ have the same k-MR $L$ according to the definition of PR3. Then we have three cases: Case (1) $aid(s) > aid(u)$; Case (2) $aid(s) = aid(u)$; Case (3) $aid(s) < aid(u)$. Case (1) can be proved by Lemma 4.7, because $aid(s) > aid(u)$, $aid(t) > aid(u)$, and both $p(s,u)$ and $p(u,t)$ have k-MR $L$. Case (2) can be proved by Lemma 4.6 because the backward KBS from $t$ can visit $u$, *i.e.*, $aid(s) = aid(u)$. The Case (3) can also by proved by 4.6 if the forward KBS from $s$ can visit $t$. The only left case is that $aid(s) < aid(u)$ and the forward KBS from $s$ cannot visit $t$ because of PR3. In this case, there must exist vertex

$v, aid(v) < aid(s) < aid(u) = n+1$, and the k-MR of $p(s,v)$ and $p(v,u)$ is $L$. Then we have $aid(v) < aid(s)$ and $aid(v) < aid(t)$, and both path $p(s,v)$ and $(v,...,u,...,t)$ have the k-MR $L$, which can be proved by Lemma 4.7.

□

Theorem 4.9. *With pruning rules, the* RLC *index is condensed.*

Proof. Assuming in the RLC index there exists index entry $(t,L) \in \mathcal{L}_{out}(s)$, and there also exist $(u,L) \in \mathcal{L}_{out}(s)$ and $(u,L) \in \mathcal{L}_{in}(t)$. Then we have $aid(u) \geq aid(t)$, otherwise $(t,L) \in \mathcal{L}_{out}(s)$ can be pruned. Given this, $(u,L) \in \mathcal{L}_{in}(t)$ can not exist because the backward KBS from $t$ performs earlier than the forward KBS from $u$, which means we have either $(t,L) \in \mathcal{L}_{out}(u)$, or $(v,L) \in \mathcal{L}_{out}(u)$ and $(v,L) \in \mathcal{L}_{in}(t)$, such that $(u,L) \in \mathcal{L}_{in}(t)$ is pruned. The proof follows the same sketch if $(s,L) \in \mathcal{L}_{in}(t)$ is considered. □

Theorem 4.10 (*Sound and Complete* RLC *Index*). *Given an edge-labeled graph $G$ and the* RLC *index of $G$ with a positive integer $k$ built by Algorithm 2. There exists a path from vertex $s$ to vertex $t$ in $G$, which satisfies a label constraint $L^+$, $|L| \leq k$, if and only if one of the following condition is satisfied*

*(1) $\exists(x,L) \in \mathcal{L}_{out}(s)$ and $\exists(x,L) \in \mathcal{L}_{in}(t)$;*
*(2) $\exists(t,L) \in \mathcal{L}_{out}(s)$, or $\exists(s,L) \in \mathcal{L}_{in}(t)$.*

Proof. (Sufficiency) It is straightforward.

(Necessity) Let $p$ be the path from $s$ to $t$ with the k-MR $L$. W.l.o.g, let the backward KBS from $t$ perform first. Then we have two cases: the backward KBS from $t$ can visit or cannot visit $s$. In the first case, the k-MR $L$ of path $p$ must be recorded according to Lemma 4.6. In the second case, PR3 must be triggered. According to Lemma 4.8, we have the k-MR of $p$ is also recorded. □

## 4.5 Complexity Analysis

*Query time.* Given a query $Q(s,t,L^+)$, the time complexity of using Algorithm 1 to answer the query is $O(|\mathcal{L}_{out}(s)| + |\mathcal{L}_{in}(t)|)$, because we only need to take $O(|\mathcal{L}_{out}(s)| + |\mathcal{L}_{in}(t)|)$ time to apply the merge join to find $(x,L) \in \mathcal{L}_{out}(s)$ and $(x,L) \in \mathcal{L}_{in}(t)$. Note that index entries in $\mathcal{L}_{out}(s)$ and $\mathcal{L}_{in}(t)$ have already been sorted according to access id of vertices, which means sorting is not required by mapping vertex id to access id during querying index.

*Indexing time.* We present the time complexity of Algorithm 2 as follows. Given a directed edge-labeled graph $G = (V,E,\mathbb{L})$, and a positive integer $k$. The worst case is that $G$ is a complete graph, and there are $|\mathbb{L}|$ edges between every pair of vertices, each of which has a distinct edge label. In this case, performing kernel-search of depth $k$ from a vertex requires $O((|\mathbb{L}||V|)^k)$, which can generate $|V|k^2$ pairs of kernel candidates and sets of frontier vertices. Thus, kernel-search from each vertex requires $O(|V|(|\mathbb{L}||V|)^k)$ time. In addition, the indexing algorithm will only perform kernel-BFS from the first vertex (the vertex with access $id = 1$, and this can be any vertex for a complete graph), instead of from each vertex in $V$. The reason is that in a complete graph, any vertex can reach any vertex through any minimum repeat of length up to $k$, such that after performing kernel-BFS from the first vertex, reachability information can be fully recorded, and the other kernel-BFSs will be pruned by PR3. Performing kernel-BFS for a kernel candidate takes $O((|V|+|E|)k)$ time. Therefore, the total time complexity of Algorithm 2 is $O(|V|^{k+1}|\mathbb{L}|^k + |V||E|k^3)$.

*Index size.* The index size can be $O(|V|^2|\mathbb{L}|^k)$, since $\mathcal{L}_{in}(v)$ or $\mathcal{L}_{out}(v)$ can contain $O(|V|C)$ index entries, where $C$ is the number of distinct minimum repeats for all label sequences derived from $|\mathbb{L}|$ of length up to $k$. It can be computed as follows, $C = \sum_{i=1}^{k} F(i)$, where $F(i) = |\mathbb{L}|^i - (\sum_{j \in fac(i), j \neq i} F(j))$ with $F(1) = |\mathbb{L}|$ and $fac(i)$ the set of factors of integer $i$. Interestingly, when the input graph is complete, where every pair of vertices has $|L|$ labels, the index size can be reduced to $C = O(|V|^2|\mathbb{L}|^k)$, because performing KBS from vertex with the first access id can fully cover the reachability information and other index entries needed to be created by the following KBS will be skipped by pruning rules.

**Table 3: Overview of real-world graphs.**

| Dataset | $\|V\|$ | $\|E\|$ | $\|\mathbb{L}\|$ | Synthetic Labels | Loop Count | Triangle Count |
|---|---|---|---|---|---|---|
| Advogato (AD) | 6K | 51K | 3 | | 4K | 98K |
| Soc-Epinions (EP) | 75K | 508K | 8 | √ | 0 | 1.6M |
| Twitter-ICWSM (TW) | 465K | 834K | 8 | √ | 0 | 38K |
| Web-NotreDame(WN) | 325K | 1.4M | 8 | √ | 27K | 8.9M |
| Web-Standford (WS) | 281K | 2M | 8 | √ | 0 | 11M |
| Web-Google (WG) | 875K | 5M | 8 | √ | 0 | 13M |
| Wiki-Talk (WT) | 2.3M | 5M | 8 | √ | 0 | 9M |
| Web-BerkStan (WB) | 685K | 7M | 8 | √ | 0 | 64M |
| Wiki-hyperlink (WH) | 1.7M | 28.5M | 8 | √ | - | 52M |
| Pokec (PR) | 1.6M | 30.6M | 8 | √ | 0 | 32M |
| StackOverflow (SO) | 2.6M | 63.4M | 3 | | 15M | 114M |
| LiveJournal (LJ) | 4.8M | 68.9M | 50 | √ | 0 | 285M |
| Wiki-link-fr (WF) | 3.3M | 123.7M | 25 | √ | 19K | 30B |

# 5 EXPERIMENTAL EVALUATION

We present experimental results in this section. We used real-world and synthetic graphs to evaluate the query time, the indexing time, and the size of the RLC index. In addition, we also compared the query time with existing systems to illustrate the benefits of our approach.

*Baseline.* Up to our knowledge, the RLC index is the first indexing technique designed for processing recursive label-concatenated reachability queries, and indices for other types of reachability queries are not usable in our context because of specific path constraints defined in the RLC queries. Thus, we compare the RLC index with BFS and Bi-BFS (bidirectional BFS) in terms of query time to understand how much improvement the index can provide against a full online traversal. In addition, we also include ETC (extended transitive closure) as a baseline. The indexing algorithm of ETC performs forward KBS from each vertex without pruning rules, and records for every pair of vertices $(u,v)$ any k-MR of any path $p(u,v)$. There are two differences between the indexing algorithm of ETC and the one of the RLC index: (1) only forward KBS is used for building ETC, instead of forward and backward KBS for the RLC index, and (2) none of the pruning rules is applied for building ETC. Finally, we also include commercial and open-sourced systems to evaluate query processing time, where our goal is to understand the query time improvement of the RLC index on top of mainstream graph data processing engines.

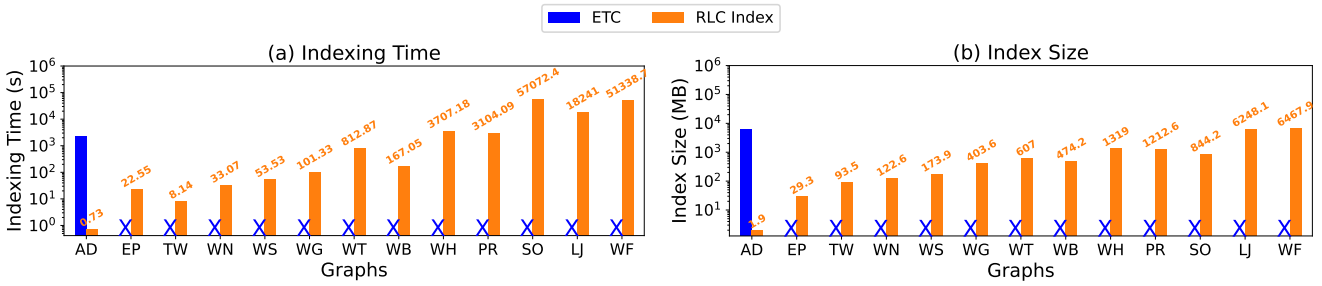


**Figure 3: Indexing time and index size of ETC and the RLC index (RI) for real-world graphs. Building ETC timed out (24 hours) for all the graphs except the AD graph.**

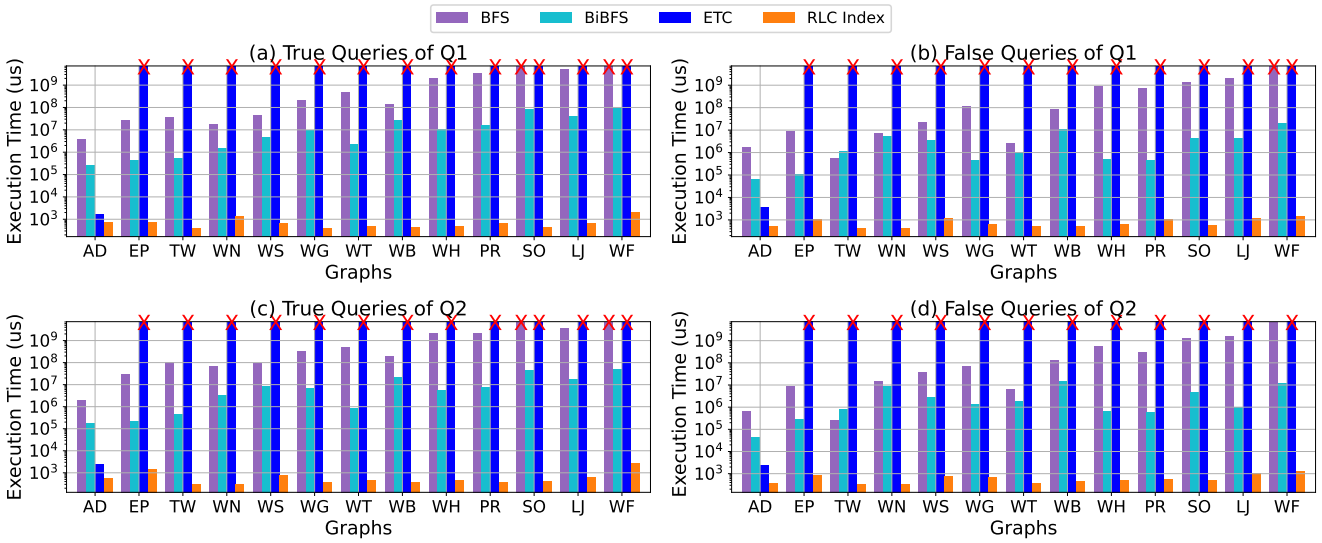


**Figure 4: Execution time of 1000-true queries or 1000-false queries of types $Q1$ or $Q2$ on real-world graphs using BFS, BiBFS, ETC and the RLC index.**

*Dataset.* We use both real-world datasets and synthetic graphs in the experiments. We present the statistics of real-world datasets in Table 3, which are from either the SNAP [29] or the KNOECT [28] project. We also include for each graph the loop count and the triangle count (cycles of length 1 and 3) shown in the last two columns in Table 3. We generate synthetic labels for graphs that do not have labels on its edges, indicated by the last column in Table 3. The edge labels are generated according to the a Zipfian distribution with exponent 2. The synthetic graphs used in our experiments follows two different modes, namely the *Erdős–Rényi* (ER) model and the *Barabási-Albert* (BA) model. ER-graphs with $|V|$ nodes and $|E|$ edges are generated by uniformly choosing a graph from all possible graphs with $|V|$ nodes and $|E|$ edges. BA-graphs with $|V|$ nodes and $|E|$ edges are generated through firstly having a complete graph of $\frac{|V|}{2000}$ nodes and then continuously adding new nodes that are connected to previously generated $\frac{|E|}{|V|}$ nodes, *e.g.*, a generated BA-graph with $1M$ nodes and $5M$ edges has a complete graph of 500 nodes. In general, ER-graphs have an almost uniform degree-distribution while BA-graphs have a skew in it. We use JGraphT [31] to generate ER- and BA-graphs in our experiments. The method to assign labels to edges in synthetic graphs is the same as the one used for real-world graphs.

**Table 4: Types of reachability queries in the experiments.**

| Name | Query | Name | Query |
|---|---|---|---|
| $Q1$ | $(a \circ b)^+$ | $Q2$ | $a^+$ |
| $Q3$ | $a \circ b^+$ | $Q4$ | $a \circ b^+ \circ c$ |
| $Q5$ | $a^+ \circ b^+$ | $Q6$ | $a \circ b^+ \circ c^+$ |

*Query type.* We use the six query types shown in Table 4, where $a$, $b$ and $c$ represent edge labels. The query types $Q1$ and $Q2$ are the main query types that can be directly supported by the RLC index, *i.e.*, using Algorithm1. These two types of queries are used in the experiments where we compare query time of the RLC index with baseline solutions (Section 5.1), and analyzing the characteristic of the index using synthetic graphs (Section 5.2). The other types of queries are additionally included in the experiments of system

comparisons (Section 5.3), and the goal is to demonstrate that the RLC index can be used to speed up the processing of a wide range of query types appearing in real-world query logs, *e.g.*, Wikidata Query Logs [11]. The way we support $Q3$, $Q4$, $Q5$, and $Q6$ in Table 4 is to use BiBfs with the RLC index. In general, for vertex pairs visited by BiBfs, the RLC index is used to check if the *true* answer can be returned, otherwise BiBfs will continue searching.

*Query generation.* For each real-word graph, we generate two query sets for $Q1$ and $Q2$ respectively, and each query set contains 1000 true-queries and 1000 false-queries. We explain the method for query generation as follows. We randomly select a source vertex $s$ and a target vertex $t$, and also randomly choose a label constraint $L^+$. Then a bidirectional breadth-first search is conducted to test whether $s$ reaches $t$ under the constraint of $L^+$. If the test returns $true$, we add $(s, t, L^+, true)$ to the true-query set, otherwise we add it into the false-query set. After that, we randomly select another $(s, t, L^+)$, and repeat the above procedure until the query set has 1000 true-queries and 1000 false-queries.

*Implementation and Setting.* Our implementation is in Java 11, including baseline solutions and the RLC index. The source codes are available online TODO: to be added, which includes every detail of the experiments, *e.g.*, datasets, query sets, method for query generation, etc. We run experiments on a machine with 8 virtual CPUs of Intel(R) Xeon(R) 2.40GHz, and 128GB main memory, where the heap size of JVM is configured to be 120GB

## 5.1 Performance on Real-World Graphs

In this section, we analyze the performance of the RLC index on real-world graphs. We compare the RLC index with ETC in terms of indexing time and index size, and with BFS and Bi-BFS in terms of query time. Each query set for each graph contains 1000 true-queries or 1000 false-queries of query types Q1 or Q2. The RLC index and ETC are built with $k = 2$ for each graph. The experimental results are presented in Fig. 3 and 4.

*Indexing time.* Building ETC cannot be completed in 24 hours for real-world graphs, as demonstrated in Fig.3 (a), with the exception of the AD graph (the smallest), which takes around 30 minutes. On the other hand, the RLC index can be efficiently built, *i.e.*, the indexing time for the first 10 graphs is at most 1 hour. The last three graphs, *i.e.*, the SO graph, the LJ graph, and the WF graph are more challenging than the others, not only because they more vertices and edges, but also because they a larger number of loops and triangles, as shown in Table 3. The SO graph has the longest indexing time due to its highly dense and cyclic character, *i.e.*, it has $15M$ loops and $114M$ triangles. Although the WF graph has much fewer loops than the SO graph, it contains $30B$ triangles. Consequently, the indexing time of the WF graph is at the same order of magnitude as the one of the SO graph. While it has more vertices and triangles than the SO graph, the LJ graph requires a lower indexing time. As a direction comparison, the RLC index for the AD graph can be built in 0.7s, leading to a four-orders-of-magnitude improvement over ETC.

*Index size.* The Fig. 3 (b) shows the size of the RLC index for real-world graphs. The size of ETC for the AD graph (the smallest) takes up around 6GB. The RLC index, on the other hand, requires much less space. Our general observation is that the graphs that require more indexing time also leads to a larger index size. In addition, the number of vertices also has an impact on index size, *e.g.*, the RLC index of the TW graph takes less time to build than the RLC index of the EP graph, but the RLC index of the TW graph takes up more space. Interestingly, the SO graph, requiring the longest indexing time, has an index size of only 844MB, which is smaller than the index size of the PR graph. However, the PR graph is much less cyclic and requires much less time to build the RLC index than the SO graph. This demonstrates that the indexing algorithm with pruning rules can effectively eliminate redundant index entries that are highly occurring in the case of a dense and cyclic graph. When compared to ETC, the size of the RLC index the AD graph is only 1.9 MB, which is a four-orders-of-magnitude improvement.

*Query time.* Query execution using BFS times out for the true-queries on both the SO graph and the WF graph, and for the false queries of type $Q1$ on the WF graph. In addition, we only report the query time of ETC for the AD graph as ETC cannot be built for all the other graphs. As demonstrated in Fig. 4, the RLC index shows a up to six- and four-orders-of-magnitude improvement against BFS and Bi-BFS, respectively, and is even slightly faster than ETC on the AD graph. More importantly, the query time of the RLC index remains almost steady, and the average time for each query is at the level of 1 microsecond, except for the WF graph (the largest) for which around 2 microseconds. We also observe that queries of type $Q1$ are slightly more difficult to process using an online traversal (BFS or BiBFS) than ones of type $Q2$, especially for true-queries on dense and cyclic graphs, such as the SO graph, the LJ graph, and the WF graph. This is because evaluating $Q1$ queries might require traversing small cycles more than once, which is not necessary for processing $Q2$ queries. In addition, false-queries are less expensive to process than true-queries as graph traversal can immediately halt if none of the outgoing edges has the desired label.

## 5.2 Impact of Graph Characteristics

In this section, we focus on analyzing the performance of the RLC index on synthetic graphs with different characteristics, namely average degree, label set size, and number of vertices. The input parameter $k$ is set to 2 for all the experiments except the one in Section 5.2.3, where we analyze the RLC index with different $k$. The synthetic graphs included in these experiments are ER-graphs and BA-graphs. There are two query sets for each graph containing 1000 true-queries and 1000 false-queries respectively, which are referred to as *ER.T* and *ER.F* for EA-graph, and *BA.T* and *BA.F* for EA-graph. Each query in a query set is of type Q1 or Q2.

*5.2.1 Impact of average degree and label set size.* In this experiment, we use BA-graphs and EA-graphs with 1 million vertices, and we vary the average degree $d$ in $(2, 3, 4, 5)$, and label set size $|\mathbb{L}|$ in $(8, 12, 16, 20, 24, 28, 32, 36)$, *e.g.*, a graph with $d = 5$ and $|\mathbb{L}| = 16$ has $1M$ vertices, $5M$ edges, and 16 distinct edge labels. We aim at analyzing indexing time, index size, and query time of the RLC index as we increase of average degree and label set size. The experimental results for ER-graphs and BA-graphs are reported in Fig. 5 and Fig. 6, respectively. We discuss the results below.

*Indexing time.* We observe that the indexing time for both ER-graphs and BA-graphs with a fixed $d$ increases as $|\mathbb{L}|$ increases. This

is because $|\mathbb{L}|$ affects indexing time complexity discussed in Section 4.5. More precisely, as $|Lab|$ increases, the number of possible minimum repeats increases, requiring more time for the kernel-search phase of KBS in the indexing algorithm to traverse the graph and generate potential kernel candidates, resulting in more kernel-BFS execution. Furthermore, because there are more edges to traverse, the indexing time for both ER-graphs and BA-graphs with a fixed $|\mathbb{L}|$ increases as $d$ increases.

*Index size.* As illustrated in Fig. 5 and 6, an increase in average degree $d$ can result in a larger index size for both ER-graphs and BA-graphs. The fundamental reason for this is that a vertex $s$ can reach a vertex $t$ through more paths, leading to a higher number of minimum repeats being recorded. As the size of the label set grows, new behaviors emerge in terms of index size. Specifically, the increase is negligible for ER-graphs with a smaller $d$, *e.g.*, 2, and becomes more noticeable for ER-graphs with a larger $d$, *e.g.*, 5. For any $d$, however, we see a noticeable increase in index size with the growth in $|\mathbb{L}|$ for BA-graphs. This is due to the fact that a BA-graph comprises a complete graph, and vertices inside the complete graph have higher degrees, consequently KBS executed from such vertices can create more index entries as $|\mathbb{L}|$ grows, as it can reach other vertices through paths with more distinct minimum repeats. However, because of the uniform degree distribution, the increase in the number of minimum repeats of paths from a vertex $s$ to a vertex $t$ due to an increase in $|\mathbb{L}|$ is not significant for ER-graphs when $d$ is small, but the corresponding impact becomes stronger when $d$ is bigger.

*Query time.* We observe in Fig. 5 and 6 that for both ER-graphs and BA-graphs, the impact of $d$ on query time is negligible. However, the growth of $|\mathbb{L}|$ has a different impact on query time. More precisely, when $|\mathbb{L}|$ rises, the execution time of both true- and false-queries for ER-graphs remains steady. When it comes to BA-graphs, increasing $|\mathbb{L}|$ can lead to a minor boost in true-query execution time but has no impact on false query execution time. This can be explained by the fact that the vertices in the complete graph of a BA-graph can reach (or be reachable from) much more vertices than the vertices outside the complete graph in the BA-graph, which can lead to a skew in the distribution of vertices in index entries, *i.e.*, many index entries have the same vertex. Furthermore, when $|\mathbb{L}|$ grows, the number of minimum repeats also increases. Therefore, the skew is higher. As a result, processing true-queries will encounter situations where the query algorithm searches for a particular minimum repeat in a significant number of index entries with the same vertex. However, for false-queries, the query result can be returned instantly if there are no index entries with the same vertex.

*5.2.2 Scalability.* In this experiment, we use BA-graphs and EA-graphs with average degree 5, 16 edge labels, and vary the number of vertices in $(125K, 250K, 500K, 1M, 2M)$. The goal is to analyze the scalability of the RLC index in terms of $|V|$. The results of indexing time, index size, and query time for both ER-graphs and BA-graphs are reported in Fig. 7.

*Indexing time and index size.* Our overall observation is that both indexing time and index size grow with the increase of the number of vertices in graphs, and the increasing rate is gradually decreasing. This can be understood as follows. Graphs with more vertices require more KBS iterations, which increases indexing time and also the number of index entries. The number of minimal repeats, on the other hand, does not increase as quickly as the growth of the number of vertices since it is mostly influenced by the average degree and label set size. Therefore, the increasing rate of indexing time and index size of ER-graphs gradually decreases. However, the degree skewing in BA-graphs will cause the number of minimum repeats to keep growing with the increase of the number of vertices, which will have an impact on the growth of indexing time and index size with the number of vertices. It is also worth noting that indexing BA-graphs is more expensive than indexing ER-graphs, because the presence of complete graphs makes BA-graphs more challenging to process.

*Query time.* In Fig. 7, for both ER-graphs and BA-graphs, we observe the phenomenon that query time increases as the number of vertices grows but the increase rate is slowing down, which is similar to indexing time and index size discussed above. In addition, for BA-graphs, true-query time is higher than false-query time. However, we see the opposite for ER-graphs. This can also be observed in Fig 5 and Fig 6, where the y-axis has the same scale. We discuss the reason as follows. Given a query $(s, t, L^+)$. When a graph has a uniform distribution in vertex degree, the index entries $(v, mr)$ in both $\mathcal{L}_{out}(s)$ and $\mathcal{L}_{in}(t)$ also have a uniform distribution in terms of $v$. Thus the query algorithm (based on merge join) tends to perform an exhaustive search in $\mathcal{L}_{out}(s)$ and $\mathcal{L}_{in}(t)$ to find index entries with the same vertex $v$, which results in false-queries taking longer time to execute than true-queries. However, when the distribution of vertex degree is skewed, the index entries in $\mathcal{L}_{out}(s)$ and $\mathcal{L}_{in}(t)$ can be dominated by a few vertices, *e.g.*, a vertex $u$ in the complete graph of a BA-graph. Furthermore, as there can exist more paths from $u$ to $s$ or $u$ to $t$, the number of index entries with $u$ can be relatively large because of distinct minimum repeats. Given this, the query algorithm can perform a faster search for false-queries than true-queries, because the number of distinct vertices in both $\mathcal{L}_{out}(s)$ and $\mathcal{L}_{in}(t)$ is not large. The query algorithm, on the other hand, needs to select a specific $mr$ among index entries with vertex $u$ of a high degree, which takes more time.

*5.2.3 Impact of $k$.* In this experiment, we aim at analyzing the impact of $k$ on the index. We use a BA-graph and a EA-graph with $125K$ vertices, average degree 5, 16 edge labels, and we build the RLC index for the BA-graph and the EA-graph with $k$ in $(2, 3, 4)$. In addition, for each graph, we evaluate each query set using the three different indices built with the three different $k$ values. The results of indexing time, index size, and query time are reported in Fig. 8.

*Overall results.* For both types of synthetic graphs, we notice that indexing time and index size rise as $k$ grows. The fundamental reason is that as $k$ increases, the number of possible minimum repeats increases exponentially, as discussed in Section 4.5. As a result, the indexing algorithm needs to search for and keep record of more minimum repeats. We also observe the query time increases when $k$ grows. This is mainly due to a larger index size since query time complexity is linear with the number of index entries. It's also worth noting that an increase in $k$ has a greater influence on BA-graph true-queries than its false-queries, and on ER-graph false-queries than its true-queries. The reason for this is that true-queries on BA-graphs (or false-queries on ER-graphs) take longer to process

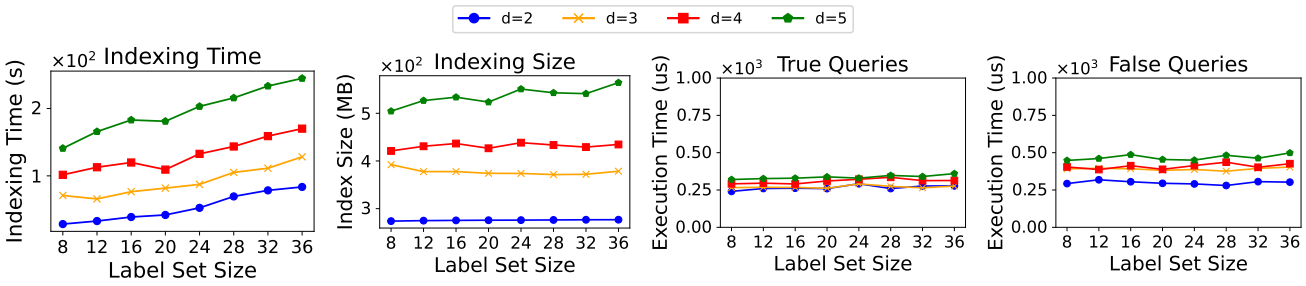


**Figure 5: Indexing time, index size, and execution time of 1000 true-queries and 1000 false-queries for ER-graphs with 1M vertices, and varying average degree $d$ and label set size.**

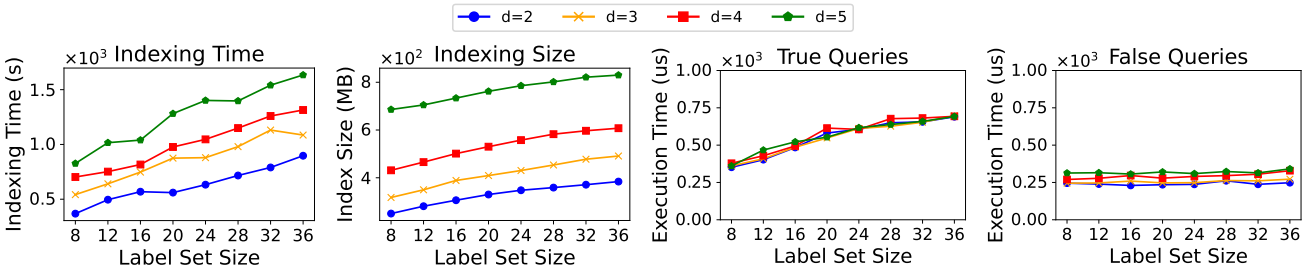


**Figure 6: Indexing time, index size, and execution time of 1000-true queries and 1000-false queries for BA-graphs with 1M vertices, and various average degree $d$ and label set size.**

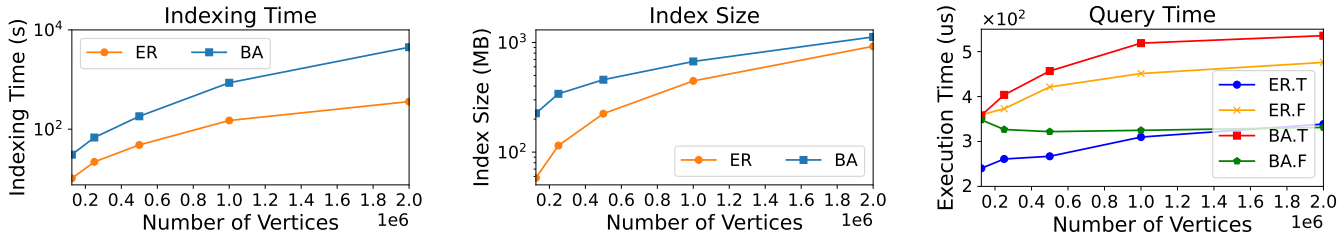


**Figure 7: Indexing time, index size, and execution time of 1000-true queries and 1000-false queries for synthetic graphs with average degree 5, 16 edge labels, and varying number of vertices.**

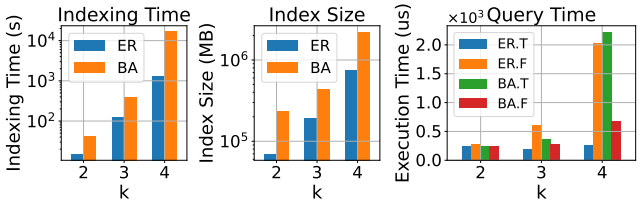


**Figure 8: Evaluation of the `RLC` index with various $k$ using BA-graphs and EA-graphs, and workloads of 1000-true queries and 1000-false queries, respectively.**

than false-queries on BA-graphs (or true-queries on ER-graphs). Furthermore, increasing $k$ can amplify the effect by causing the exponential rise of minimal repeats.

## 5.3 Speedup on graph engines

TODO: graphs: WN and SO

TODO: The discussion will focus on the following two arguments: (1) for Q1 and Q2, index can speed up query execution; (2) meanwhile, index can also speed up other types of queries

TODO: We will not discuss the performance differences between these systems.

TODO: breaking points to be added.

# 6 RELATED WORK

## 6.1 Plain Reachability

Given an unlabeled graph $G = (V, E)$ and a pair of vertices $(s, t)$, a plain reachability query asks whether there exists a path from $s$ to $t$. The existing approaches lie between two extremes, *i.e.*, online

**Table 5: Speed-up of the RLC index on the WN graph.**

| Anon. Sys. | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 |
|---|---|---|---|---|---|---|
| Sys1 | 2.4e+3 | 6.11e+3 | 2.92e+3 | 5.7e+3 | 2.23e+4 | 1.6e+4 |
| Sys2 | 3.59e+4 | 6.47e+4 | 3.13e+4 | 9.62e+3 | 1.74e+5 | 1.93e+5 |
| Sys3 | - | 5.73e+2 | - | - | - | - |

**Table 6: Speed-up of the RLC index on the SO graph.**

| Anon. Sys. | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 |
|---|---|---|---|---|---|---|
| Sys1 | 6.13e+5 | 1.63e+5 | 4.95e+6 | 6.11e+4 | – | – |
| Sys2 | - | 9.67e+6 | 2.19e+6 | 1.74e+4 | 7.33e+6 | 3.52e+4 |
| Sys3 | | | | | | |

traversal and transitive closure, and try to find a good balance between query time, indexing time and index size. We briefly review the literature, whereas comprehensive surveys can be found in [10, 37, 47]. They mainly fall into two main categories [37, 43]: *(1)* Index-Only approaches; *(2)* Index+Graph approaches. The former can answer queries using only the index, while the latter requires online graph traversal, which can be guided by auxiliary data, *i.e.*, partial index, and can deal with large graphs. The Index-Only approaches can be further classified into two sub-categories: *(1.1)* cover-based approaches, which use simple graph structures to cover an input graph, such as *Chain Cover* [13, 22], *Tree Cover* [5], and *Path-Tree Cover* [26]; *(1.2)* hop-based approaches, which decompose the path between a reachable pair of vertices into sub-paths passing through intermediate vertices, such as 2-Hop labeling [16] and 3-Hop labeling [25]. As the minimal *2-Hop* cover problem is NP-hard, various approaches have been proposed for improving the index construction of *2-Hop* labeling, *e.g.*, *TF-Label* [15], *HL* [24], *DL* [24], and *TOL* [48]. The Index+Graph approaches can fall into two sub-categories: *(2.1)* position-based approaches, *e.g.*, *Tree+SSPI* [12], *GRIPP* [39], *GRAIL* [46], and *Ferrari* [36]; *(2.2)* set-containment-based approaches, *e.g.*, *IP* [43] and *BFL* [37].

RLC queries are different from plain reachability queries because they are evaluated on labeled graphs and due to additional recursive label-concatenated constraint. Therefore, the approaches used to evaluate plain reachability queries are not applicable to RLC queries. More precisely, indexing techniques for plain reachability queries only record information about graph structure but ignore information of edge labels.

## 6.2 Recursive Label-Alternated Queries

Recursive label-alternated queries are reachability queries with a path constraint that is based on alternation of edge labels (instead of concatenation as in our work), which are known as LCR queries in the literature. LCR queries have been extensively studied in the last decade. We provide an overview for this line of works below.

Jin *et. al* [23] presented the first result on LCR queries. To compress the generalized transitive closure that records reachable pairs and sets of path-labels, the authors proposed sufficient label sets and a spanning tree with partial transitive closure. The main idea is to record only the minimal label sets for paths with non-tree edges as the starting and ending edge in a partial transitive closure, then the generalized transitive closure can be recovered by traversing the spanning tree and looking up the partial transitive closure. The Zou *et. al* [49] method finds all strongly connected components (SCCs) in an input graph, and replaces each SCC with a bipartite graph to obtain an edge labeled DAG. Zou *et. al* proposed an efficient algorithm to compute generalized transitive closures for the DAG using its topological order. To handle a large graph, the algorithm is applied to graph partitions instead of SCCs. Valstar *et. al* [40] proposed a landmark-based index. In a nutshell, the method computes the generalized transitive closure for a subset of vertices called *landmarks* that have the highest total degree, and applies online BFS to answer LCR queries that can be accelerated by hitting landmarks. The method is also optimized by adding a fixed number of index entries for non-landmark vertices, such that the search space for negative queries can be pruned. The state-of-the-art indexing techniques for LCR queries are the Peng *et. al* [35] method and the Chen *et. al* [14] method. Peng *et. al* [35] proposes the LC 2-hop labeling, which extends 2-hop labeling framework through adding minimal sets of path-labels for each entry in the 2-hop labeling. Chen *et. al* [14] proposes a recursive method to handle LCR queries, where an input graph is recursively decomposed into spanning trees and graph summaries, and LCR queries are decomposed into sub-queries evaluated using spanning trees and graph summaries.

LCR queries are similar to RLC queries in the sense that path-label information is mandatory to check reachability. Their major difference is on the type of the regular expression given in queries. The expression in LCR queries is an alternation of edge labels, while the one in RLC queries is a concatenation of edge labels. The completely different path constraint makes LCR indices infeasible for processing RLC queries. More precisely, LCR indices store label sets for a pair of vertices $(s, t)$, such a set is not sufficient to answer an RLC query with $(s, t)$ because the order and the number of occurrences of edge labels are missing, *i.e.*, the stored index entries are label sets, instead of label sequences required by processing RLC queries. The difference in path constraint also makes indexing algorithms for LCR queries and RLC queries fundamentally different. More precisely, for an LCR indexing algorithm, it is sufficient to traverse any cycle in a graph only once. Then based on a snapshot of an LCR index, any further traversal along the cycle can be skipped because the reachability information related to the cycle under an alternation-based path constraint has already been recorded. However, it is not true for the case of RLC queries, where a cycle (particularly a loop) might need to be traversed multiple times depending on label sequences to be checked along paths. Therefore, indexing approaches for LCR queries are not applicable to indexing RLC queries. In our work, we show how to properly adapt the successful 2-hop labeling framework [6, 16] to the design and efficient deployment of RLC Index.

## 6.3 Regular Reachability Queries

Regular reachability queries correspond to reachability queries with path constraints specified using regular expressions, *i.e.*, checking the existence of a path based on the regular expressions. Under the simple path semantics, it is NP-complete to evaluate regular reachability queries [30]. By restricting regular expressions or graph

instances, there exist tractable cases under this semantics [8, 30]. When it comes to the arbitrary path semantics, regular reachability queries can be processed by using automata-based techniques [9, 44], bi-directional BFS [18], partial evaluation for distributed graphs [19], incremental approach for streaming graphs [33, 34]. Compared to these solutions, we focus on designing an index-based solution to handle reachability queries with a concatenation of edge labels constraint under the arbitrary path semantics, since such queries are computationally hard to process due to in-depth graph traversals. To the best of our knowledge, our work is the first of its kind focusing on the design of a reachability index for such queries.

## 7 CONCLUSION

TODO: to be added ...
  TODO: parallelization of indexing, billion-scale graphs ...

## REFERENCES


[1] [n.d.]. Apache Jena. https://jena.apache.org.
[2] [n.d.]. Graph Query Language GQL Standard. https://www.gqlstandards.org/.
[3] [n.d.]. Neo4j. http://www.opencypher.org.
[4] [n.d.]. Virtuoso. http://vos.openlinksw.com/owiki/wiki/VOS.
[5] R. Agrawal, A. Borgida, and H. V. Jagadish. 1989. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/67544.66950
[6] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 349–360. https://doi.org/10.1145/2463676.2465315
[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages. https://doi.org/10.1145/3104031
[8] Guillaume Bagan, Angela Bonifati, and Benoit Groz. 2013. A Trichotomy for Regular Simple Path Queries on Graphs. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (New York, New York, USA) *(PODS '13)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/2463664.2467795
[9] Pablo Barceló Baeza. 2013. Querying Graph Databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (New York, New York, USA) *(PODS '13)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/2463664.2465216
[10] Angela Bonifati, George Fletcher, Hannes Voigt, Nikolay Yakovets, and H. V. Jagadish. 2018. *Querying Graphs*. Morgan & Claypool Publishers.
[11] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 127–138. https://doi.org/10.1145/3308558.3313472
[12] Li Chen, Amarnath Gupta, and M. Erdem Kurul. 2005. Stack-Based Algorithms for Pattern Matching on DAGs. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 493–504.
[13] Y. Chen and Y. Chen. 2008. An Efficient Algorithm for Answering Graph Reachability Queries. In *2008 IEEE 24th International Conference on Data Engineering*. 893–902. https://doi.org/10.1109/ICDE.2008.4497498
[14] Yangjun Chen and Gagandeep Singh. 2021. Graph Indexing for Efficient Evaluation of Label-Constrained Reachability Queries. *ACM Trans. Database Syst.* (2021).
[15] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: A Topological-Folding Labeling Scheme for Reachability Querying in a Large Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/2463676.2465286
[16] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and Distance Queries via 2-Hop Labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California) *(SODA '02)*. Society for Industrial and Applied Mathematics, USA, 937–946.
[17] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *ArXiv* abs/1901.08248 (2019).
[18] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. 2011. Adding regular expressions to graph reachability and pattern queries. In *2011 IEEE 27th International Conference on Data Engineering*. 39–50. https://doi.org/10.1109/ICDE.2011.5767858
[19] Wenfei Fan, Xin Wang, and Yinghui Wu. 2012. Performance Guarantees for Distributed Reachability Queries. *Proc. VLDB Endow.* 5, 11 (July 2012), 1304–1316. https://doi.org/10.14778/2350229.2350248
[20] Haixun Wang, Hao He, Jun Yang, P. S. Yu, and J. X. Yu. 2006. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *22nd International Conference on Data Engineering (ICDE'06)*. 75–75. https://doi.org/10.1109/ICDE.2006.53
[21] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. 2015. PGX.D: a fast distributed graph processing engine. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807620
[22] H. V. Jagadish. 1990. A Compression Technique to Materialize Transitive Closure. *ACM Trans. Database Syst.* 15, 4 (Dec. 1990), 558–598. https://doi.org/10.1145/99935.99944
[23] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. 2010. Computing Label-Constraint Reachability in Graph Databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/1807167.1807183
[24] Ruoming Jin and Guan Wang. 2013. Simple, Fast, and Scalable Reachability Oracle. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1978–1989. https://doi.org/10.14778/2556549.2556578
[25] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-HOP: A High-Compression Indexing Scheme for Reachability Query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 813–826. https://doi.org/10.1145/1559845.1559930
[26] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently Answering Reachability Queries on Very Large Directed Graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 595–608. https://doi.org/10.1145/1376616.1376677
[27] D. Knuth, James H. Morris, and V. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6 (1977), 323–350.
[28] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173
[29] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[30] A. O. Mendelzon and P. T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *Proceedings of the 15th International Conference on Very Large Data Bases* (Amsterdam, The Netherlands) *(VLDB '89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 185–193.
[31] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V. Sichi. 2020. JGraphT—A Java Library for Graph Data Structures and Algorithms. *ACM Trans. Math. Softw.* 46, 2, Article 16 (may 2020), 29 pages. https://doi.org/10.1145/3381449
[32] Mark E. J. Newman. 2010. *Networks: An Introduction*. Oxford University Press.
[33] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1415–1430. https://doi.org/10.1145/3318464.3389733
[34] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2021. Evaluating Complex Queries on Streaming Graphs. arXiv:2101.12305 [cs.DB]
[35] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering Billion-Scale Label-Constrained Reachability Queries within Microsecond. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 812–825. https://doi.org/10.14778/3380750.3380753
[36] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. 2013. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1009–1020. https://doi.org/10.1109/ICDE.2013.6544893
[37] J. Su, Q. Zhu, H. Wei, and J. X. Yu. 2017. Reachability Querying: Can It Be Even Faster? *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (2017), 683–697. https://doi.org/10.1109/TKDE.2016.2631160
[38] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. 2021. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 209–224. https://www.usenix.org/conference/atc21/presentation/trigonakis

[39] Silke Trißl and Ulf Leser. 2007. Fast and Practical Indexing and Querying of Very Large Graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 845–856. https://doi.org/10.1145/1247480.1247573

[40] Lucien D.J. Valstar, George H.L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 345–358. https://doi.org/10.1145/3035918.3035955

[41] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (Redwood Shores, California) *(GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2960414.2960421

[42] R. R. Veloso, Loïc Cerf, W. Meira, and Mohammed J. Zaki. 2014. Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach. In *EDBT*.

[43] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2014. Reachability Querying: An Independent Permutation Labeling Approach. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1191–1202. https://doi.org/10.14778/2732977.2732992

[44] Peter T. Wood. 2012. Query Languages for Graph Databases. *SIGMOD Rec.* 41, 1 (April 2012), 50–60. https://doi.org/10.1145/2206869.2206879

[45] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast and Scalable Reachability Queries on Graphs by Pruned Labeling with Landmarks and Paths. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management* (San Francisco, California, USA) *(CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 1601–1606. https://doi.org/10.1145/2505515.2505724

[46] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2010. GRAIL: Scalable Reachability Index for Large Graphs. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 276–284. https://doi.org/10.14778/1920841.1920879

[47] Jeffrey Xu Yu and Jiefeng Cheng. 2010. Graph reachability queries: A survey. In *Managing and Mining Graph Data*. Springer, 181–215.

[48] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability Queries on Large Dynamic Graphs: A Total Order Approach. In *Proceedings of the 2014 ACM International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1323–1334. https://doi.org/10.1145/2588555.2612181

[49] Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. 2014. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems* 40 (2014), 47–66. https://doi.org/10.1016/j.is.2013.10.003