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 used to check the existence of a path from a source node to a target node are fundamental operators for querying and processing graph data. In this paper, we study the case of processing recursive label-concatenated reachability queries, referred to as RLC queries. These queries check the existence of a path that can satisfy the constraint defined by a concatenation of at most k edge labels under the Kleene plus. Such reachability queries are becoming important in practical network analysis and graph database applications. We have found that current graph database engines cannot efficiently support RLC queries, and little research has been conducted on them. We present the RLC index, the first reachability index for processing these queries. The query algorithm of the RLC index checks whether the source vertex can reach an intermediate vertex that can also reach the target vertex under the path constraint. We propose an indexing algorithm to build the RLC index, which guarantees the soundness and the completeness of query execution and avoids recording redundant index entries. Comprehensive experiments on real graphs show that the RLC index can significantly reduce both the offline processing cost and the memory overhead of the transitive closure while improving query processing up to six orders of magnitude over online traversals.

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 https://github.com/g-rpqs/rlc-index.

1 INTRODUCTION

Graphs have been the natural choice of data representation in various domains [35], *e.g.,* social, biochemical, and transportation networks, where the primary focus is to discover the relationships

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

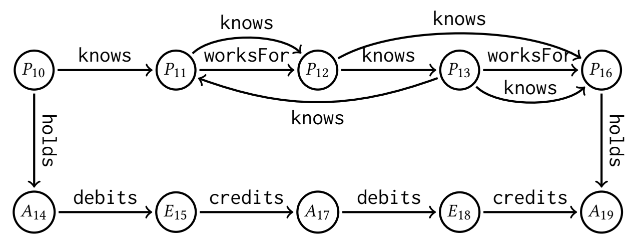

Figure 1: An edge-labeled graph with nodes of persons (P), accounts (A), and external entities (E), and five edge labels: knows, worksFor, holds, debits, and credits.

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 [6, 14, 15, 17, 18, 22, 24, 26–28, 40, 41, 43, 46, 47, 50].

To facilitate the representation of different types of relationships in real-world applications, *edge-labeled graphs* (or *property graphs*), where single labels can be assigned to edges, are more widely adopted than the simple form of a graph. 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) [8, 12]. Concatenation and the Kleene plus (or star) are two fundamental operators to form a regular expression in RPQs. In this work, we consider reachability queries with a path constraint of the Kleene plus over a concatenation of edge labels, which are referred to 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 a 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 $Q1(A_{14}, A_{19}, (\texttt{debits}, \texttt{credits})^{+})$ 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, 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 [13], which is the largest repository of open-source graph queries (of the order of 500M graph queries). In particular, RLC queries often time out in these logs [13] thus showing the limitations of graph query engines to efficiently evaluate them. Moreover, Neo4j(v4.3) [3] and TigerGraph(v3.3) [19], two of the mainstream 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 [45] supported by Oracle PGX [23, 42], and in SPARQL 1.1 (ASK query) supported by Virtuoso [5], Apache Jena [1], etc. Currently, 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 an index-based efficient evaluation of RLC queries. Indexing is one of the open problems for future graph processing systems, as it permits to improve and predict the performance of graph queries [39]. Presently, most of the indexing methods have been focused on alternation of labels instead of concatenation [38, 44] and are not able to evaluate 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 [18]. More precisely, in the RLC index, we assign to each vertex v two sets $\mathcal{L}_{in}(v)$ and $\mathcal{L}_{out}(v)$ of pairs of vertices and sequences, where $\mathcal{L}_{in}(v)$ contains vertices u that can reach v and the (sub-)sequences of edge labels from u to v, and $\mathcal{L}_{out}(v)$ contains vertices w that are reachable from v and the (sub-)sequences of edge labels from v to w. Then, the RLC index processes an 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. We found that k usually has an upper bound in practice. As an example, in massive real-world query logs [13], 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, timed out in the logs, as they are complex to evaluate (NP-complete under the simple path semantics [10, 33]). 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, the indexing algorithm for building the RLC index conducting backward and forward BFS from each vertex. 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 the size of the RLC index.

Contribution. Our main contributions are summarized as follows:

- We introduce RLC queries, reachability queries with the path constraint of the Kleene plus over a concatenation of edge labels. 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 traversals.
- We propose the RLC index, the first reachability index for processing RLC queries, and a corresponding indexing algorithm. We formally prove that the constructed RLC index is sound and complete for a parameter k given arbitrarily, where redundant index entries can be completely removed.
- Our comprehensive experiments using real-world graphs show that the RLC index can be efficiently built and can significantly reduce the memory overhead of the (extended) transitive closure. Meanwhile, the RLC index is capable of answering RLC queries efficiently, up to six orders of magnitude over online traversals. We also demonstrate the speed-up and the generality of using the RLC index to accelerate query processing on mainstream graph systems.

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 in Section 4. Experimental results are presented in Section 5. We discuss related works in Section 6 and conclude in Section 7. All related proofs are included in our technical report, which is available online [4].

2 PROBLEM STATEMENT

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

In this paper, we consider the *arbitrary paths* semantics [8], *i.e.*, allowing for duplicate vertices along the path. 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 $v_0, ..., v_n \in V$, $e_1, ..., e_n \in E$, 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 sequence of edge labels of a path from u to v.

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 ϵ the empty label sequence, *i.e.*, $|\epsilon| = 0$. We use $\circ$ to denote the concatenation of label sequences (or labels), *i.e.*, $(l_1, ..., l_i) \circ (l_{i+1}, ..., l_n) = (l_1, ..., l_n)$, or $L \circ L = L^2$. For the empty label sequence ϵ, we define $L \circ \epsilon = \epsilon \circ L = L$.

The label sequence $L' = (l'_1, ..., l'_{n'})$, $|L'| = n'$ 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 the 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 $MR(p(P_{10}, P_{16}))$ = (`knows,worksFor`). If $L = MR(L)$, we also say L itself 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 $MR(L)$.

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

Suppose L has two minimum repeats, then only the shorter is $MR(L)$. Therefore, we have Lemma 2.1.

2.2 `RLC` Query

In this work, we consider the path constraint $L^+ = (l_1, ..., l_k)^+$, where '+' is the Kleene plus, *i.e.*, one-or-more concatenations of the label sequence $L = (l_1, ..., l_k)$. We focus on a path-constraint L, s.t. $L = MR(L)$, *e.g.*, L^+ = (`knows,worksFor`)$^+$. 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 Query). Given an edge-labeled directed graph $G = (V, E, \mathbb{L})$ and a constant k, an `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*.

Notice that for the sake of simplicity in this paper we focus on the `RLC` queries with the Kleene plus. Queries (s, t, L^*) with the Kleene star can be trivially reduced to queries with the Kleene plus (s, t, L^+) through checking whether s is equal to t. Our method is thus applicable to `RLC` queries with the Kleene star. It is also worth noting that path queries that do not belong to the above fragment, *i.e.*, with L^+ s.t. $L \neq MR(L)$ impose an additional constraint on the length of paths boiling down to the even-path problem (NP-complete [10, 31, 33] for simple paths). For these reasons, these queries with additional constraints are not considered further in our work.

Given an `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 the transitive closure, where for each pair of vertices (s, t) we record whether $s \rightsquigarrow t$ and all the label sequences from s to t. Note that the naive approach to build the transitive closure is not usable in our case because of cycles on the path from s to t. We adopt an *extended transitive closure*, which is presented in Section 5. However, as demonstrated in our experiments, these two solutions require either too much query time or storage space, which are impractical for a large graph.

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 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

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 first observe that recording MRs for a pair of vertices, 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 `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.*, $S^k(s, t) = \{L | p \in \mathbb{P}(s, t), MR(\Lambda(p)) = L, |L| \leq k\}$.

To deal with `RLC` queries, we need to compute and record the concise label sequences. We have Proposition 2.4 by definition.

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. In general, 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 the subsequent search.

Example 3.1 (Illustrating Example). Consider the graph in Fig. 1. Assume we need to compute $S^2(P_{11}, P_{13})$, *i.e.*, $k = 2$, and we perform a breadth-first search from P_{11}. 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 search depth reaches $2k = 4$, *i.e.*, P_{12} is visited for the second time, we can have 4 different label sequences, which are L_1 = (`knows, knows, knows, knows`), L_2 = (`knows, knows, knows, worksFor`), L_3 = (`worksFor, knows, knows, knows`), and L_4 = (`worksFor, knows, knows, worksFor`). Given this, all the 4 label sequences except L_1 do not need to be expanded anymore, because their expansions cannot produce a minimum repeat whose length is not larger than 2. After this, the following search continued with L_1 is guided by $(\texttt{knows})^+$ that is computed from L_1. 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 ϵ 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 ϵ.

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.*, $(\texttt{knows})^+$ 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 3.3 that KBS can compute a sound and complete $S^k(s,t)$. The proof of Theorem 3.3 is included in our technical report [4] due to the space limit.

THEOREM 3.3. *Given a path p from u to v and a positive integer k, 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.*

In Case 3 of Theorem 3.3, if $\Lambda(u,x)$ of p does not have a kernel, then p does not have a non-empty k-MR. In another sense, Theorem 3.3 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 a length of $2k$ from u. In addition, with kernels determined in Theorem 3.3, we will not miss any k-MRs for paths from u to v.

We discuss below two strategies to compute kernels based on Theorem 3.3, 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 3.3 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 require 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 the subsequent search. As KBS is a breadth-first search, the set of kernel candidates does not miss any valid kernels. Although an invalid kernel may be included, the search guided by the invalid kernel will not reach a target vertex through a path of which the k-MR is the invalid kernel. Therefore, the set of concise label sequences computed by the eager strategy is still sound and complete.

Example 3.4. Consider the example of computing $S^2(P_{10}, P_{13})$ in Fig. 1. Using the eager strategy, when P_{12} is visited for the first time, two kernel candidates can be determined , *i.e.*, (`knows`) and (`knows, worksFor`). Although an invalid kernel (`knows, worksFor`) is included, the search guided by $(\texttt{knows, worksFor})^+$ cannot reach P_{13} through a path that has the (`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 label sequences of length $2k$ from a source vertex is more expensive than the case of paths of length k, especially on a dense graph. In addition, as kernel candidates can be used to guide the search in the phase of kernel-BFS, vertices can be marked to avoid redundant traversal, which is not possible in the kernel-search phase aiming at exploring all possible label sequences.

4 RLC INDEX

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

4.1 Overarching Idea

Given an RLC query (s, t, L^+), $|L| \leq k$, the idea is to check whether there exists a 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 the 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

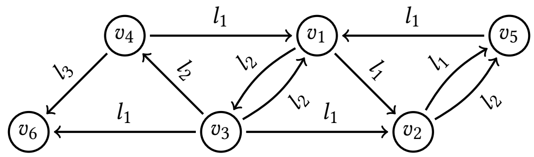

Figure 2: Running example.

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 the 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 $Q_3(v_1, v_3, (l_1)^+)$, 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 problem when $\mathbb{L}$ contains only one label, and finding the minimum 2-hop labeling is NP-hard [18]. Although it is expensive to find the minimum RLC index, it is still worthwhile to remove as many redundant index entries as possible. The main idea 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.

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 entries $(u, L') \in \mathcal{L}_{out}(s)$ 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 an 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 Algorithm 1. The output of the merge join is a set of index entry pairs (I', I''), such that $I'.vid = I''.vid$. 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.

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)	-

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. Suppose the backward KBS from vertex v_i is visiting v. If $|MR(\Lambda(v, v_i))| \leq k$, then we add $(v_i, MR(\Lambda(v, v_i)))$ into $\mathcal{L}_{out}(v)$. Although there may be cycles in a graph, the KBS will not go on forever, because when the depth of the search reaches k, the KBS will be transformed from its kernel-search phase into its kernel-BFS phase that is guided by the Kleene plus of a kernel candidate, such that the KBS can terminate if any invalid label (or state) transition is met, or a vertex being visited with a label l_i of the kernel has been visited with a label l_j of the kernel, s.t. $i = j$.

KBSs are executed from each vertex in V and the 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 for the 2-hop labeling frameworks [7, 38, 49]. The id of vertex v in the sorted list is referred to as the access id of v, 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 below the backward KBS from v_1, which is the first search of the indexing algorithm. 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

Algorithm 1: Query Algorithm

```
procedure Query(s, t, L⁺)
    if ∃(t, L) ∈ ℒ_out(s) or ∃(s, L) ∈ ℒ_in(t) then
        return true;
    for (I′, I″) ∈ mergeJoin(ℒ_out(s), ℒ_in(t)) do
        if I′.mr = L and I″.mr = L then
            return true;
    return false;
```

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 the backward kernel-BFS 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 from v_1 guided by $(l_1)^+$, but this kernel-BFS will not continue when it visits v_5 that has already been visited with the label l_1.

4.3.2 Pruning Rules. To accelerate index construction as well as remove redundant index entries, we apply pruning rules during KBSs. For ease of presentation, we present the pruning rules for backward KBSs, and the same rules apply 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:*** *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 all the vertices in $in(v_i)$ are 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 4.7 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

Algorithm 2: Indexing Algorithm.

```
procedure kernelBasedSearch(v, k)
    for (L, vSet) ∈ backwardKernelSearch(v, k) do
        backwardKernelBFS(v, vSet, L);
    for (L, vSet) ∈ forwardKernelSearch(v, k) do
        forwardKernelBFS(v, vSet, L);
procedure backwardKernelSearch(v, k)
    q ← an empty queue of (vertex, label sequence);
    q.enqueue(v, ε);
    map ← a map of (kernel candidates, vertex set);
    while q is not empty do
        (x, seq) ← q.dequeue();
        for in-coming edge e(y, x) to x do
            seq′ ← λ(e(y, x)) ∘ seq; L ← MR(seq′);
            insert (y, v, L); map.get(L).add(x);
            if |seq′| < k then
                q.enqueue(y, seq′);
    return map;
procedure insert(s, t, L)
    if aid(t) > aid(s) or Query(s, t, L⁺) then // PR 2 or 1
        return false;
    else
        add (t, L) into ℒ_out(s);
        return true;
procedure backwardKernelBFS(v, vSet, L)
    q ← an empty queue of (vertex, integer);
    for x ∈ vSet do
        mark x as visited by state 1, q.enqueue(x, |L|);
    while q is not empty do
        (x, i) ← q.dequeue(), i ← i − 1;
        if i = 0 then i = |L| ;
        label l ← L.get(i);
        for in-coming edge e(y, x) to x do
            if l ≠ λ(e(y, x)) or y was visited by state i then
                continue;
            if i = 1 and insert(y, v, L) then        // PR 3
                continue;
            q.enqueue(y, i); mark y visited by state i;
```

$(v_3, (l_2, l_1))$ 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 the kernel-search phase to the kernel-BFS phase that is guided by $(l_2, l_1)^+$. When v_2 visits itself for the first time, the KBS tries to create index entry $(v_2, (l_2, l_1)) \in \mathcal{L}_{in}(v_2)$. However, this index entry can be pruned by

PR1 because of $(v_1, (l_2, l_1)) \in \mathcal{L}_{out}(v_2)$ and $(v_1, (l_2, l_1)) \in (v_2)$. In this case, PR3 is also triggered, which means v_2 and all the vertices in $out(v_2)$ are 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 [29] 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 35, 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 an RLC query, *i.e.*, in Case 1 of Definition 4.1, L' can be different from L'' in the alternative 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 the version introduced in this paper, *e.g.*, even for the smallest graph used in our experiments (the AD graph presented in Section 5), the indexing time of the alternative version 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 4.6 that pruning rules can guarantee the condensed property of the RLC index, and in Theorem 4.7 that the RLC index constructed by Algorithm 2 is correct, *i.e.*, sound and complete. The proofs of Theorem 4.6 and Theorem 4.7 are included in our technical report [4] due to the space limit.

THEOREM 4.6. *With pruning rules, the* RLC *index is condensed.*

THEOREM 4.7. *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 $\exists(s, L) \in \mathcal{L}_{in}(t)$.

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 the access id of vertices, such that we do not need to sort index entries when applying the merge join.

Index size. The index size can be $O(|V|^2|\mathbb{L}|^k)$, since each $\mathcal{L}_{in}(v)$ or $\mathcal{L}_{out}(v)$ can contain $O(|V|C)$ index entries, where $C = O(|\mathbb{L}|^k)$ is the number of distinct minimum repeats for all label sequences derived from $|\mathbb{L}|$ of length up to k. C can be computed as follows, $C = \sum_{i=1}^{k} F(i)$, where $F(i) = |\mathbb{L}|^i - (\sum_{j \in f(i), j \neq i} F(j))$ with $F(1) = |\mathbb{L}|$ and $f(i)$ the set of factors of integer i.

Indexing time. In Algorithm 2, we perform a KBS from each vertex, and each KBS consists of two phases: the kernel-search phase and the kernel-BFS phase. Performing a kernel-search of depth k from a vertex requires $O(|\mathbb{L}|^k|V|^k)$ time, and generates $O(|L|^k)$ kernel candidates as discussed in the index size analysis. Each kernel candidate requires a kernel-BFS taking $O(|E|k)$ time. Hence the time complexity for performing a KBS is $O(|\mathbb{L}|^k|V|^k + |L|^k|E|k)$. Then the total time complexity is $O(|V|^{k+1}|\mathbb{L}|^k + |L|^k|V||E|k)$.

5 EXPERIMENTAL EVALUATION

In this section, we study the performances of the RLC index. We used real-world and synthetic graphs to evaluate the indexing time and the size of the RLC index, and we focused on evaluating the execution time of RLC queries using the RLC index. In addition, we compared the query time of the RLC index with existing systems, where we additionally consider more types of reachability queries from real-world query logs to show the generality of our approach.

Baselines. To the best of 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, the chosen baselines for the RLC index are BFS (breadth-first search) and BiBFS (bidirectional BFS), which we used to understand in terms of query time how much improvement our index can provide against a full online traversal. In addition, we also include an extended transitive closure as a baseline, which is referred to as ETC. The indexing algorithm of ETC performs a forward KBS from each vertex without pruning rules, and records for every reachable pair of vertices (u, v) any k-MR of any path $p(u, v)$. In ETC, we use a hashmap to store reachable pairs of vertices and the corresponding set of k-MRs. 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.

Datasets. We use both real-world datasets and synthetic graphs in our experiments. We present the statistics of real-world datasets in Table 3, which are from either the SNAP [32] or the KNOECT [30] project. We also include for each graph the loop count (cycles of length 1) and the triangle count (cycles of length 3) shown in the last two columns in Table 3. We generate synthetic labels for graphs that do not have labels on their edges, indicated by the column of synthetic labels in Table 3. The edge labels have been generated according to the Zipfian distribution [9] 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 first 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

Table 3: Overview of real-world graphs. In this table, graphs are sorted according to the number of their edges in ascending order.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathbb{L}\rvert$	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-Stanford (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	√	4K	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

Table 4: Indexing time (IT) and index size (IS) for graphs in Table 3. In this table, "-" indicates that the method timed out on the graph.

Dataset	RLC Index		ETC	
	IT (s)	IS (MB)	IT (s)	IS (MB)
AD	0.7	1.9	2216.1	2798.7
EP	22.6	29.3	-	-
TW	8.1	93.5	-	-
WN	33.1	122.6	-	-
WS	53.5	173.9	-	-
WG	101.3	403.6	-	-
WT	812.9	607.1	-	-
WB	167.1	474.2	-	-
WH	3707.2	1319.1	-	-
PR	3104.1	1212.6	-	-
SO	57072.5	844.2	-	-
LJ	18240.9	6248.1	-	-
WF	51338.7	6467.9	-	-

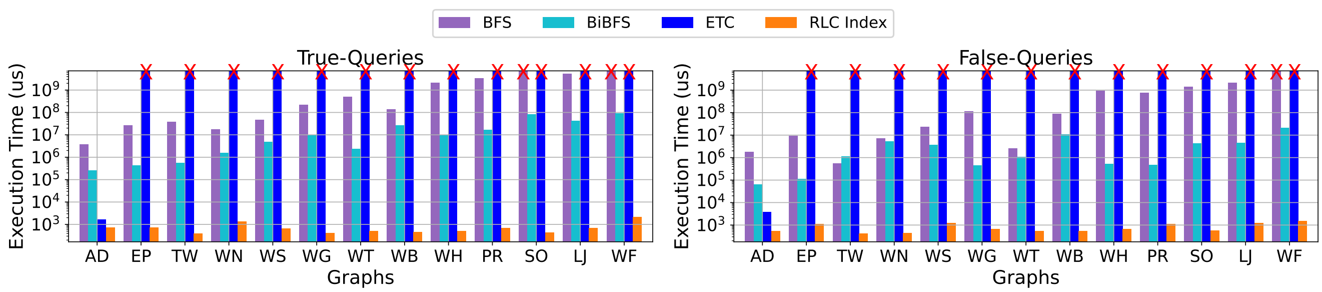

Figure 3: Execution time of 1000-true queries on real-world graphs using BFS, BiBFS, ETC and the RLC index.

1M nodes and 5M edges has a complete sub-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 [34] to generate the ER- and BA-graphs. The method to assign labels to edges in synthetic graphs is the same as the one used for real-world graphs.

Query generation. For each real-word graph, we generate two query sets, each of which contains 1000 true-queries and 1000 false-queries, respectively. Each query in the generated query sets has the expression of $(a \circ b)^+$, where a and b represent two distinct edge labels, and the concatenation of a and b is under the Kleene plus. We explain the method for query generation as follows. We uniformly select a source vertex s and a target vertex t, and also uniformly 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 generate another (s, t, L^+), and repeat the above procedure until the completion of the two query sets.

Implementation and Setting. Our implementation has been done in Java 11, spanning baseline solutions and the RLC index. The source code[1] is accessible online, as well as the datasets and query workloads. 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 BiBFS in terms of query time. The RLC index and ETC are built with a parameter $k = 2$ for each graph.

Indexing time. Table 4 shows the indexing time of the RLC index and ETC on real-world graphs. Building ETC cannot be completed in 24 hours for real-world graphs (or it runs out of memory) except for the AD graph with the least number of edges. The RLC index for the AD graph can be built in 0.7s, leading to a four-orders-of-magnitude improvement over ETC. The indexing time improvement of the RLC index over ETC mainly comes from the application of the pruning rules that can skip vertices in KBSs when building the RLC index. The indexing time of the RLC index 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 have more vertices and edges, but also because they have a larger number of loops and triangles, as shown in Table 3. The SO graph has the longest indexing time due to its highly

[1]Open Source Link: https://github.com/g-rpqs/rlc-index

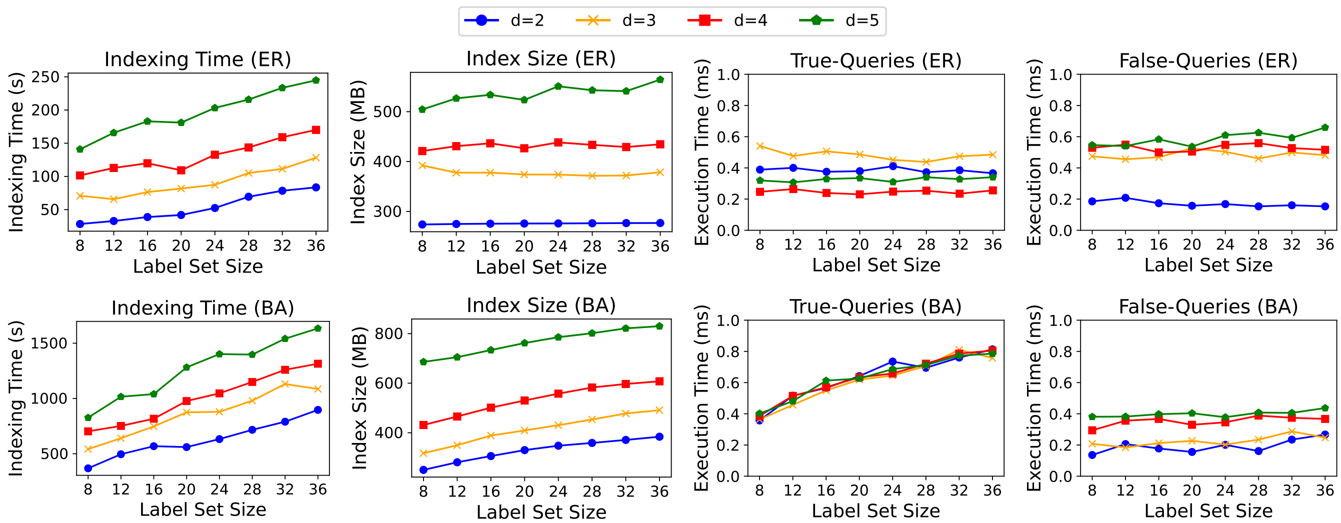

Figure 4: Indexing time, index size, and execution time of 1000 true-queries and 1000 false-queries for ER-graphs and BA-graphs with 1M vertices, and varying average degree d and label set size.

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.

Index size. Table 4 shows the size of the RLC index for real-world graphs. The size of the RLC index is much smaller than the size of ETC for the AD graph (that is the only graph ETC can be built within 24 hours). The index size improvement of the RLC index over ETC is not only because of the adoption of the 2-hop labeling framework in RLC but also the application of pruning rules that can avoid recording redundant index entries. The effectiveness of pruning rules can also be observed between the PR graph and the SO graph. Although the SO graph is larger than the PR graph in terms of the number of vertices and the number of edges, the index size of the SO graph is smaller than the index size of the PR graph.

Query time. Fig. 3 shows the execution time of the true-query set and the false-query set. In general, the execution time of a query set of 1000 queries using the RLC index is around 1 millisecond for all the graphs in Table 3, except for the WF graph (that has the largest number of edges) for which around 2 milliseconds. Query execution using BFS times out for the true-queries on both the SO graph and the WF graph, and for the false queries 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. 3, the RLC index shows an up to six-orders-of-magnitude improvement over BFS and four-orders-of-magnitude improvement over BiBFS. The query time using the RLC index is slightly faster than ETC, because the larger number of reachable pairs of vertices recorded in ETC leads to a slight overhead of checking for the reachability between a source s and a target t and also the existence of a minimum repeat of a path from s to t.

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, the number of vertices, and the input parameter k. The synthetic graphs included in these experiments are ER-graphs and BA-graphs. We generate for each graph a query set of 1000 true-queries and a query set of 1000 false-queries, which are referred to as *ER.T* and *ER.F* for an EA-graph, and *BA.T* and *BA.F* for a BA-graph. The input parameter k is set to two for all experiments but one in Section 5.2.3, where we analyze the RLC index with different k.

5.2.1 Impact of label set size and average degree. In this experiment, we use BA-graphs and EA-graphs with 1M 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 the increase of average degree and label set size. The experimental results for ER-graphs and BA-graphs are reported in Fig. 4. We discuss the results below.

Indexing time. We observe in Fig. 4 that the indexing time for the used ER-graphs and BA-graphs with a fixed d shows a linear increase (for most cases) as $|\mathbb{L}|$ increases. This can be understood as follows. When $|\mathbb{L}|$ increases, the number of possible minimum repeats increases, requiring more time for the kernel-search phase of a KBS in the indexing algorithm to traverse the graph and generate potential kernel candidates, resulting in more kernel-BFS executions. The total number of possible minimum repeats can be quadratic in $|\mathbb{L}|$ in the worst case when the input parameter k of the RLC index is two. Furthermore, because there are more edges to traverse, the indexing time for both ER-graphs and BA-graphs with a fixed $|\mathbb{L}|$ increases linearly as d increases.

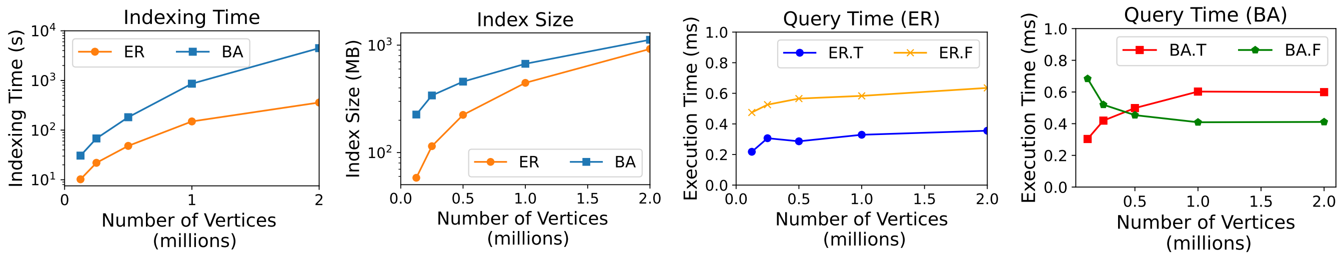

Figure 5: Indexing time, index size, and execution time of 1000-true queries and 1000-false queries for synthetic graphs with average degree 5, 16 edge labels, and varying number of vertices.

Index size. As illustrated in Fig. 4, an increase in average degree d can result in a larger index size for both ER-graphs and BA-graphs. The reason 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, the corresponding impact on ER-graphs is different from the one on BA-graphs. Specifically, the increase is negligible for ER-graphs with a small d, *e.g.*, 2, and becomes more noticeable for ER-graphs with a large d, *e.g.*, 5. For any d, however, we see a clear linear increase in index size with the growth in $|\mathbb{L}|$ for BA-graphs. This is because a BA-graph comprises a complete sub-graph, and vertices inside the complete sub-graph have higher degrees. Therefore, the KBSs 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 larger.

Query time. As shown in Fig. 4 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 almost no impact on false query execution time. This can be explained by the fact that the vertices in the complete sub-graph of a BA-graph can reach (or be reachable from) much more vertices than the vertices outside the complete sub-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. Fig. 4 also shows that d has a negligible impact on the execution time of the true-queries on the BA graph. A possible explanation for this might be that the number of index entries for some vertices in the BA graph does not increase significantly w.r.t the increase of d because of the skew in the degree distribution and a fixed $|\mathbb{L}|$.

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. 5.

Indexing time and index size. Fig. 5 shows that both indexing time and index size grow sub-linearly with the increase of the number of vertices in graphs. 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 the indexing time and the 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 a complete sub-graph makes BA-graphs more challenging to process.

Query time. Fig. 5 shows that the false-query time on the ER-graphs and the true-query time on the BA-graphs increase sub-linearly as the number of vertices grows. In addition, the false-query time is higher than the true-query time on ER-graphs, and the true-query time is higher than the false-query time on BA-graphs. We interpret the results 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 several vertices, *e.g.*, a vertex u in the complete sub-graph of a BA-graph. Furthermore, as there might 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 to process true-queries, which takes more time.

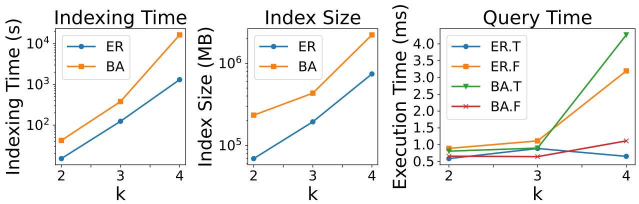

Figure 6: Evaluation of the RLC index with varying k using BA-graphs and EA-graphs, and workloads of 1000-true queries and 1000-false queries, respectively.

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 an 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 a true-query set of 1000 queries and a false-query set of 1000 queries 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. 6.

Overall results. Fig. 6 shows that the indexing time and index size of both types of synthetic graphs rise exponentially 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 records of more minimum repeats. The exponential increase of minimum repeats w.r.t k also has an impact on query execution time, particularly on the true-queries of BA-graphs and the false-queries on ER-graphs. This is mainly due to larger index size, and the true-queries on BA-graphs and the false-queries on ER-graphs are more expensive to process than their opposites, respectively.

5.3 Comparison with existing systems

In this section, we focus on evaluating how much improvement the RLC index can provide over mainstream graph processing systems. We recall that many current graph query engines fail to evaluate RLC queries, thus we focused on three systems that are able to evaluate these queries on property graphs and RDF graphs. In order to show how the index performs with varying types of queries, such as longer concatenations or path queries frequently occurring in real-world query logs, we consider the following queries: **Q1** being a single label under the Kleene plus a^+; **Q2** consisting of a concatenation of length 2 $(a \circ b)^+$; **Q3** having the expression $(a \circ b \circ c)^+$ thus a concatenation of length 3. In this case, we build the index with $k = 3$ to support all the three RLC queries, especially Q3 having the longest concatenation. For the sake of completeness, an extended reachability query with the constraint $a^+ \circ b^+$ is also included in this experiment, which is referred to as **Q4**. We have employed Q4 to study the generality and applicability of the RLC index to a wide range of regular path queries in real-world graph query logs [13]. To deal with this query, we use the RLC index in combination with an online traversal to continuously check whether intermediately visited vertices can satisfy the path constraint.

Three graph engines, including commercial and open-sourced ones, are used in the experiments, and their names are anonymized and referred to as Sys1, Sys2, and Sys3 in the results. For the systems that support multi-threaded query evaluation, we set the system to single-threaded to ensure a fair comparison with our approach, which is single-threaded only. We use the WN graph as a representative of real-world graphs, which has a moderate number of vertices and edges, along with a sufficient number of cycles to evaluate. We build one RLC index with the parameter $k = 3$ for the WN graph and use it to process all four queries. In this case, the RLC index of the WN graph can be built in 5.9 minutes and takes up 821 megabytes. We run each query using each system within the 10-minutes time limit, and we repeat the execution 20 times and report the median of the query execution time. We use two metrics to evaluate the improvement of the RLC index, namely speed-ups (SU) and the workload size break-even points (BEP). SU shows the query time improvement of the RLC index over an included graph system. BEP indicates the number of queries that make the indexing time of the RLC index pay off.

Table 5: Speed-ups (SU) and workload size break-even points (BEP) of the RLC index over graph engines on the WN graph. In this table, "-" indicates that the query execution of this system timed out.

Anon. Sys.	RLC Query						Extended Query	
	Q1		Q2		Q3		Q4	
	SU	BEP	SU	BEP	SU	BEP	SU	BEP
Sys1	1200x	84100	10400x	34000	18400x	9400	34000x	300
Sys2	3000x	34900	202000x	1700	1300000x	130	104000x	98
Sys3	597x	180000	4900x	71700	38100000x	5	-	-

The experimental results are reported in Table 5. The RLC index shows that using a single index lookup can gain significant speed-ups over included systems for processing Q3, which has the longest concatenation under the Kleene plus. We use the BEP value to understand the amortized cost of using the RLC index to accelerate Q3 processing. In particular, executing Q3 130 times on Sys2 is equivalent to the time it takes to build and query the RLC index the same number of times. The RLC index can also significantly improve the execution time for Q1, Q2, and Q4. To summarize, given a workload, an RLC index should be built with k being the length of the longest concatenation under the Kleene plus in that workload. The RLC index provide significant speedup even for queries where the concatenation length under the Kleene plus is less than k.

6 RELATED WORK

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 traversals and the 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 [12, 41, 51]. They mainly fall into two categories [41, 47]: *(1)* Index-Only approaches; *(2)* Index+Graph approaches. The former can answer queries using only the index, while the latter requires an 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 a graph, such as *Chain Cover* [15, 24], *Tree Cover* [6], and *Path-Tree Cover* [28]; *(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 [18] and 3-hop labeling [27]. 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* [17], *HL* [26], *DL* [26], and *TOL* [52]. The Index+Graph approaches can fall into two sub-categories: *(2.1)* position-based approaches, *e.g.*, *Tree+SSPI* [14], *GRIPP* [43], *GRAIL* [50], and *Ferrari* [40]; *(2.2)* set-containment-based approaches, *e.g.*, *IP* [47] and *BFL* [41].

RLC queries are different from plain reachability queries because they are evaluated on labeled graphs to find the existence of a path satisfying additional recursive label-concatenated constraints. Thus, 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.

Recursive Label-Alternated Queries. Recursive label-alternated queries are reachability queries with a path constraint that is based on an 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 of this line of works below.

Jin *et. al* [25] 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* [53] 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* [44] 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 an 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* [38] method and the Chen *et. al* [16] method. Peng *et. al* [38] proposes the LC 2-hop labeling, which extends the 2-hop labeling framework through adding minimal sets of path-labels for each entry in the 2-hop labeling. Chen *et. al* [16] 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 in 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 [7, 18] to the design and efficient deployment of the RLC index.

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 regular expressions. Under the simple path semantics, it is NP-complete to evaluate regular reachability queries [33]. By restricting regular expressions or graph instances, there exist tractable cases under this semantics [10, 33]. When it comes to the arbitrary path semantics, regular reachability queries can be processed by using automata-based techniques [11, 48], bi-directional BFS [20], partial evaluation for distributed graphs [21], incremental approach for streaming graphs [36, 37]. Compared to these solutions, we focus on designing an index-based solution to handle reachability queries with a recursive 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

In this paper, we introduced RLC queries, reachability queries with a path constraint based on the Kleene plus over a concatenation of edge labels. RLC queries are becoming important in real-world applications, such as analyzing social networks and financial transactions. In order to efficiently process RLC queries online, we propose the RLC index, the first reachability index based on the 2-hop labeling framework. We design an indexing algorithm with pruning rules to not only accelerate the index construction but also remove redundant index entries. Our comprehensive experimental study demonstrates that the RLC index strikes a good balance between online computation (full online traversals) and offline computation (a materialized transitive closure). Future research directions of this work will focus on (i) extending the RLC index to support the inverse operator of edge labels in path constraints, *e.g.*, Two-way Regular Path Queries, and (ii) parallelizing the indexing algorithm.

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.]. The technical report of the RLC index. https://github.com/g-rpqs/rlc-index/tree/main/paper/technical-report.pdf.

[5] [n.d.]. Virtuoso. http://vos.openlinksw.com/owiki/wiki/VOS.

[6] 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

[7] 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

[8] 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

[9] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Transactions on Knowledge and Data Engineering* 29, 4 (2017), 856–869.

[10] 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

[11] 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

[12] Angela Bonifati, George Fletcher, Hannes Voigt, Nikolay Yakovets, and H. V. Jagadish. 2018. *Querying Graphs*. Morgan & Claypool Publishers.

[13] 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

[14] 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.

[15] 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

[16] Yangjun Chen and Gagandeep Singh. 2021. Graph Indexing for Efficient Evaluation of Label-Constrained Reachability Queries. *ACM Trans. Database Syst.* (2021).

[17] 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

[18] 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.

[19] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *ArXiv* abs/1901.08248 (2019).

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] D. Knuth, James H. Morris, and V. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6 (1977), 323–350.

[30] 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

[31] Andrea S. LaPaugh and Christos H. Papadimitriou. 1984. The even-path problem for graphs and digraphs. *Networks* 14 (1984), 507–513.

[32] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[33] 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.

[34] 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

[35] Mark E. J. Newman. 2010. *Networks: An Introduction*. Oxford University Press.

[36] 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

[37] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2021. Evaluating Complex Queries on Streaming Graphs. arXiv:2101.12305 [cs.DB]

[38] 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

[39] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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*.

[47] 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

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

[49] 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

[50] 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

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

[52] 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

[53] 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

A PROOF OF THEOREM 3.3

A proof of Theorem 3.3 is provided in this section. Before that, we first show the kernel of a label sequence is unique in Lemma A.1 that will be used in the proof of Theorem 3.3.

Lemma A.1. *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 prove 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 $\bar{L}$ also 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 ϵ, or a proper prefix of L_3. Therefore, $\bar{L}$ 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. □

Based on Lemma A.1, we prove Theorem 3.3 below.

Proof of Theorem 3.3. 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. Assume that p can have a non-empty k-MR L''' in this case. Because $|L'''| \leq k$ and $|\Lambda(u,x)| = 2k$, then $\Lambda(u,x)$ has a kernel and a tail, which is contradiction to the case definition.
- Case (ii): $\Lambda(u,x)$ has a kernel L' and a tail L'', but $MR(L'' \circ \Lambda(x,v)) \neq L'$. Assume that p has a non-empty k-MR L''' in this case. Knowing that $|L'''| \leq k$ and $|\Lambda(u,x)| = 2k$, we have $L''' = L'$ because the kernel of $\Lambda(u,x)$ is unique (Lemma A.1). 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.

□

B PROOF OF THEOREM 4.6 AND 4.7

We prove Theorem 4.6 and Theorem 4.7 in this section. Before proceeding further, we first present the following lemmas that will be used to prove the two theorems.

Lemma B.1. *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 B.2. *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 the 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, 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 B.3. *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. There exists two cases: $aid(t) \le aid(s)$ or $aid(t) > aid(s)$. We prove the case of $aid(t) \le aid(s)$. 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 $aid(u) \ge aid(t)$ 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 B.2, 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 B.1 because the backward KBS from t can visit u, *i.e.*, $aid(s) = aid(u)$. Case (3) can also by proved by B.1 if the forward KBS from s can visit t. The only case left 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 B.2. □

Proof of Theorem 4.6. Assuming there exists index entry $(t,L) \in \mathcal{L}_{out}(s)$ in the RLC index , and there also exist $(u,L) \in \mathcal{L}_{out}(s)$ and $(u,L) \in \mathcal{L}_{in}(t)$. Then we have $aid(u) \ge aid(t)$, otherwise $(t,L) \in \mathcal{L}_{out}(s)$ can be pruned. Given this, $(u,L) \in \mathcal{L}_{in}(t)$ cannot exist because the backward KBS from t is performed 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. □

Proof of Theorem 4.7. (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 be performed 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 B.1. In the second case, PR3 must be triggered. According to Lemma B.3, we have the k-MR of p is also recorded. □