DLCR : Efficient Indexing for Label-Constrained Reachability Queries on Large Dynamic Graphs

Xin Chen
The Chinese University of Hong Kong
xchen@se.cuhk.edu.hk

You Peng*
The Chinese University of Hong Kong
ypeng@se.cuhk.edu.hk

Sibo Wang
The Chinese University of Hong Kong
swang@se.cuhk.edu.hk

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

ABSTRACT

Many real-world graphs, e.g., social networks, biological networks, knowledge graphs, naturally come with edge-labels, with different labels representing different relationships between nodes. On such edge-labeled graphs, a fundamental query is the *label-constrained reachability (LCR)* query, where we are given a source s, a target t, a label set Ψ, and the goal is to determine if there exists any path from s to t such that for any edge on the path the label belongs to Ψ. The existing indexing scheme for LCR queries still focuses on static graphs, despite the fact that many edge-labeled graphs are dynamic in nature.

Motivated by the limitations of existing solutions, we present a study on how to effectively maintain the indexing scheme on dynamic graphs. Our proposed approach is based on the state-of-the-art 2-hop index for LCR queries. In this paper, we present efficient algorithms for updating the index structure in response to dynamic edge insertions/deletions and demonstrate the correctness of our update algorithms. Following that, we present that adopting a query-friendly but update-unfriendly indexing scheme results in surprisingly superb query/update efficiency and outperforms those update-friendly ones. We analyze and demonstrate that the query-friendly indexing scheme actually achieves the same time complexity as those of update-friendly ones. Finally, we present the batched update algorithms where the updates may include multiple edge insertions/deletions. Extensive experiments show the effectiveness of the proposed update algorithms, query-friendly indexing scheme, and batched update algorithms.

PVLDB Reference Format:
Xin Chen, You Peng, Sibo Wang, and Jeffrey Xu Yu. DLCR : Efficient Indexing for Label-Constrained Reachability Queries on Large Dynamic Graphs. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Graph is a fundamental data structure that captures complicated connections between entities. Many real-world graphs, e.g., social networks, biological networks, and knowledge graphs, are edge-labeled, where each edge is associated with a label and different labels indicate different relationships. For instance, on social networks, the relationships between two users could take in a variety of forms, e.g., "Follow", "Like", and "friendOf". On such edge-labeled graphs, a fundamental type of graph query is the *label-constrained reachability (LCR)* query. In an LCR query, it takes as input a source node s, a target node t, and a label set Ψ. The query then returns true if there exists a path P from s to t such that the label λ of each edge e on path P belongs to Ψ, and otherwise returns false. As shown in [12, 19, 25], LCR queries find many applications on social networks, biological networks, knowledge graphs, etc. For instance, on a social network, an LCR query can be used to determine if two vertices are related via a series of provided relationships. Another example of the application is on knowledge graphs. Regular path queries have been extensively explored on knowledge graphs [3, 4, 26] and are supported by practical graph query languages such as SPARQL 1.1, PGQL [24], and openCypher[10]. LCR queries are one of the most important operators in regular path queries.

In the above applications, graphs are usually dynamically changing. For example, on social networks, two nodes may make new connections or interactions. On knowledge graphs, new relationships may be identified between two nodes during knowledge harvesting. However, the state-of-the-art indexing scheme *P2H+* [19] proposed by Peng et al. is based on the 2-hop index, and assumes that the input graph is static. When the graph has changed, the P2H+ index no longer works. Computing the P2H+ index from scratch after every update is not a sound option since the index construction still takes quite high pre-computational costs. An alternative solution is to do a graph traversal, e.g., BFS/DFS, and verify if there exists any path that fulfills the label constraints. However, labeled graphs in real-world applications tend to be enormous, making it expensive to answer an LCR query with online graph traversal. To avoid the expensive online traversal, ARRIVAL [25] offers an index-free, sampling-based algorithm that works for large dynamic graphs. Nevertheless, ARRIVAL could only provide an approximate result, and provides theoretical guarantees only when the input graph is strongly connected, while many labeled graphs usually include hierarchies and are not strongly connected. In addition, the query time of ARRIVAL is nearly 1000X slower than P2H+. These limit the applications of the approximate ARRIVAL.

Main Contributions. Motivated by the limitations of existing solutions, we investigate how to design an indexing scheme on dynamic graphs that is both efficient and scalable, while providing

*You Peng is the corresponding author and joint first author.

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

exact query results. Our solution is based on the state-of-the-art P2H+ 2-hop index [19]. Instead of computing the index from scratch, a set of affected nodes are effectively computed and then updates are only processed on the index structures of these affected nodes caused by the edge insertion/deletion. Since the graph is dynamically changing, a natural idea to maintain the index structure is to adopt an update efficient data structure like RB-tree. However, we demonstrate that surprisingly, both the query and update operations benefit from maintaining a query-friendly but update-unfriendly data structure, e.g., by a sorted dynamically sizable array. To explain, even in the updates, query needs to be performed first, and the query efficiency becomes the major bottleneck; therefore, the key to making the update efficient turns out to query efficiency. This discovery may also shed insights on how to design effective dynamic index structures with the 2-hop indexing scheme for other queries as well. Theoretical analysis demonstrates that the query-friendly design achieves the same time complexity as those with update-friendly data structures. Finally, updates may come in batches, and we further devise batch update algorithms to ensure that when collective updates can benefit, our update algorithm improves the update efficiency. To summarize, our principal contributions are as follows.

- *Dynamic LCR index (DLCR).* To our best knowledge, this is the first work to investigate a fully dynamic algorithm for LCR problem in an efficient manner on large graphs. We design update algorithms so that it will only need to update a small portion of affected nodes, making it super-efficient.
- *Query-friendly design.* We further investigate how to make updates more efficient considering the choice of the underlying data structure. We show that a query-friendly design actually achieves significant improvement over an update-friendly one.
- *Batch Updates.* Additionally, batch insertion and deletion techniques are proposed to further enhance performance.
- *Efficiency and Effectiveness.* Our comprehensive experiments demonstrate the effectiveness and efficiency of our algorithms comparing to baselines.

2 PRELIMINARY

This section first presents the problem definition, followed by the discussion of the state-of-the-art indexing method. Table 1 summarizes the key notations used throughout this paper.

2.1 Problem Definition

The input graph $G = (V, E, \Lambda)$ is an edge-labeled directed graph where V is a set of n vertices, Λ is a finite non-empty set of labels, and $E \subseteq V \times V \times \Lambda$ is a set of directed labeled edges. For example, $e = \langle u, v, \lambda \rangle \in E$ is an edge from u to v with label λ. A path P from s to t in graph G is a sequence of edges $\langle e_0, e_1, ..., e_k \rangle$ where $e_i = \langle u_i, u_{i+1}, \lambda_i \rangle$, $u_i \in V$, $e_i \in E$ for every $i \in [0, k]$, $u_0 = s$, and $u_{k+1} = t$. We denote the length of path P, i.e., the number k of edges on P as $|P|$. Furthermore, we say that P is an Ψ-*path* if for edge e_i on path P, its label $\lambda_i \in \Psi$. In this paper, we say that a node s can reach another node t through the label set Ψ, denoted by $s \overset{\Psi}{\leadsto} t$, if there is such a Ψ-path from s to t. Otherwise, we say s cannot reach t through the label set Ψ, denoted by $s \overset{\Psi}{\not\leadsto} t$. The label constrained reachability query is defined as follows.

Table 1: Frequently used notations

Notations	Definitions
Λ	The set of labels in the input graph
$s \overset{\Psi}{\leadsto} v$	s can reach v via label set Ψ
$s \xrightarrow{\lambda} t$ or $\langle s, t, \lambda \rangle$	An edge $\langle s, t \rangle$ with label λ on the edge
$L_{in}(v)$, $L_{out}(v)$	The in and out 2-hop index entries of node v
$InvL_{in}(v)$, $InvL_{out}(v)$	The inverted in and out 2-hop index entries of v
$rank(v), rank^{-1}(i)$	The rank of node v, the node with rank i
$\langle s, \Psi, \langle u, v, \lambda \rangle \rangle$	An entry with source id s, label set Ψ and last edge $\langle u, v, \lambda \rangle$
$\mathcal{A}_f$ ($\mathcal{A}_b$)	Forward (backward) affected node
$\mathcal{P}_f$ ($\mathcal{P}_b$)	Forward (backward) skipped path

Definition 1 (Label constrained reachability query). *Given a source node s, a target node t, and a label set Ψ, a label constrained reachability (LCR) query returns true if there exists a Ψ-path from s to t and returns false otherwise.*

In addition, we say that a label set $\Psi \subseteq \Lambda$ is a *minimal* label set connecting s to t if *(i)* $s \overset{\Psi}{\leadsto} t$ and *(ii)* $s \overset{\Psi'}{\not\leadsto} t$ for any label set $\Psi' \subsetneq \Psi$.

2.2 2-Hop Cover Framework

The 2-hop cover technique has been extensively studied in the literature (e.g., [1, 2, 6, 7]). Our method follows the same framework. In the 2-hop index for LCR queries, two sets of index entries $L_{in}(v)$ and $L_{out}(v)$, denoted as the *in-entry set* and *out-entry set* of vertex v respectively, are maintained for each vertex v. For the in-entry set $L_{in}(v)$, it stores a set of tuples in form of $\langle s, \Psi_s, e_v \rangle$, which corresponds to a Ψ_s-path P from s to v and e_v is the last edge connecting to v on path P. For the out-entry set $L_{out}(v)$, it stores a set of tuples in the same form. Assume that an entry $\langle t, \Psi_t, e_v \rangle$ is in $L_{out}(v)$, then it indicates a Ψ_t-path from v to t and the starting edge of P from v is e_v. We use a placeholder $\cdot$, e.g., $\langle t, \Psi_t, \cdot \rangle$, if the corresponding information in the tuple does not affect in the context. In addition, we use $s \overset{\Psi}{\leadsto} t$ to represent a path corresponding to an entry.

2-hop cover. In the 2-hop indexing scheme for LCR queries, an important property is the 2-hop cover. To explain, if s can reach t through label set Ψ, then we can always find a node v such that there exists an entry $\langle v, \Psi_{in}, \cdot \rangle$ in the in-entry set $L_{in}(t)$ of t and an entry $\langle v, \Psi_{out}, \cdot \rangle$ in the out-entry set $L_{out}(s)$ of node s such that $\Psi_{in} \subseteq \Psi$ and $\Psi_{out} \subseteq \Psi$. Since the out-entry $\langle v, \Psi_{out}, \cdot, \rangle$ from s indicates a Ψ_{out}-path from s to v and an in-entry $\langle v, \Psi_{in}, \cdot, \rangle$ indicates a Ψ_{in}-path from v to t. It indicates that there exists a $(\Psi_{in} \cup \Psi_{out})$-path from s to t. Since $\Psi_{in} \subseteq \Psi$, $\Psi_{out} \subseteq \Psi$, we know there exists a Ψ-path from s to t, indicating that s can reach t through label set Ψ. In this case, we also say v Ψ-*covers* s to t.

Given such a property, to answer an LCR query with input source node s, target node t, and label set Ψ, it can simply return true if a node v exists in both $L_{out}(s)$ and $L_{in}(t)$ such that v Ψ-covers s to t and otherwise return false.

We say that a 2-hop cover to be a *minimum 2-hop cover* if deleting any entry will cause the incorrect LCR query answer for some queries, i.e., some queries will not be covered by the entries after deletion. Note that not necessarily all 2-hop indices are minimum since we can simply add more entries without affecting the correctness of the queries. Next, we first will elaborate on the label-constrained BFS, followed by the state-of-the-art 2-hop indexing scheme Pruned 2-Hop (P2H+for short) for LCR queries proposed by Peng et al. [19].

2.3 Existing solutions for LCR Queries

LC-BFS. We first explain how label constrained breadth-first search (LC-BFS) works as the state-of-the-art P2H+ index is constructed in an iterative manner by LC-BFS with pruning strategies. Given a source s, the LC-BFS works as follows. Instead of maintaining a queue, it maintains $|\Lambda| + 1$ queues $Q_0, Q_1, Q_1, Q_2, \cdots, Q_{|\Lambda|}$ to track the paths from s to visited nodes and their corresponding label set, where queue Q_i records those visited nodes by a label set with a size i. Like how BFS tracks paths, we maintain the last edge of a path to encode the path information. For each node v, instead of maintaining a visit mark (as can be done for typical BFS), it maintains a set $\mathcal{S}_v$ of pairs $\langle \Psi, e_v \rangle$ indicating that for any label set $\Psi \in \mathcal{S}$, s can Ψ-reach v through the last edge e_v. Initially, S_v is set to empty for every node v. Next, $\langle s, \emptyset, NULL \rangle$ is added into Q_0 and other queues are initialized as empty. During the BFS traversal, in each iteration, it always de-queues a path from the queue with the smallest label set size. Assume that the path P is de-queued from queue Q_i, then it indicates that Q_0 to Q_{i-1} are empty. Further assume that the de-queued path P ends at node v with a label set Ψ_v. Then, in this iteration, it examines if there exists any entry $\langle \Psi, \cdot \rangle \in \mathcal{S}_v$ such that $\Psi_v \subseteq \Psi$. If this is the case, the path P is pruned and the iteration finishes. Otherwise, an entry $\langle \Psi, \cdot \rangle$ is added to S_v first. Next, it examines each out-going edge $e_w = (v, w, \lambda)$ of v and set the label set to be $\Psi_w = \Psi_v \cup \{\lambda\}$. For path P' going through P and edge e_w, it is then added into the queue $Q_{|\Psi_w|}$ and the iteration finishes. The LC-BFS terminates when all queues are empty.

The above algorithms can be used to answer the LCR query with s as the source node (with any choice of the target node and any choice of the label set). If the target node and the label set Ψ are also given, the algorithm terminates as soon as we find a path P that can connect s to t through a label set $\Psi' \subseteq \Psi$.

The time complexity of the LC-BFS can be bounded by $O(2^{|\Lambda|} \cdot (n + m))$. To explain, for each node v, there are $O(2^{|\Lambda|})$ different label sets that are not a subset of each other in the worst case. Thus we explore each node and each edge at most $2^{|\Lambda|}$ times.

Example 1. *Consider an input graph G_1 as shown in Figure 1(a). The set Λ of labels is $\{a, b\}$ and $|\Lambda| = 2$. Thus, we maintain 3 queues: Q_0, Q_1, and Q_2. Assume that we conduct an LC-BFS from node 1. Initially, for each node v, we maintain a set S_v initialized as an empty set, add $\langle 1, \emptyset, \cdot \rangle$ into Q_0, and initialize Q_1 to Q_2 to be empty.*

In the first iteration, $\langle 1, \emptyset, \cdot \rangle$ is de-queued from Q_0. Since S_1 is empty, $\langle \emptyset, \cdot \rangle$ is added to S_1. Next, following the outgoing edge $\langle 1, 3, a \rangle$ of node 1, we obtain a path P_1 from 1 to 3 via label set $\{a\}$ and add P_1 to Q_1. In the next iteration, path P_1 is de-queued from Q_1 as Q_1 is the non-empty queue with the smallest id. Since S_3 is empty, $\langle \{a\}, \cdot \rangle$ is added to S_3. Then the edges $\langle 3, 4, a \rangle$ and $\langle 3, 4, b \rangle$ are visited. Two paths P_2 from node 1 to 4 via label set $\{a\}$ and path P_3 from node 1 to 4 via label set $\{a, b\}$ are derived. Then, we add P_2 to Q_1 and P_3 to Q_2.

Next, path P_2 from node 1 to 4 via label set $\{a\}$ is de-queued from Q_1. Since S_4 is empty, an entry $\langle \{a\}, \cdot \rangle$ is added to S_4. Since node 4 has no out-neighbors, the iteration finishes. After that, path P_3 from node 1 to 4 via label set $\{a, b\}$ is de-queued from Q_2 since Q_0 and Q_1 are empty. As S_4 include an entry $\langle \{a\}, \cdot \rangle$ such that $\{a\} \subseteq \{a, b\}$, path P_3 is pruned and the current iteration finishes. After this iteration, all queues are empty and the LC-BFS from node 1 terminates. □

P2H+ index. The P2H+ index is a 2-hop index that is built according to *vertex ranks*, whose definition is as follows.

Definition 2 (Vertex Rank). *Assume that the vertex ids have been mapped to 1 to n where n is the number of nodes in the graph. Let rank be a bijection from $\{1, 2, \cdots, n\}$ to $\{1, 2, \cdots, n\}$. Then the vertex rank of node v is $rank(v)$.*

There are many ways to define vertex ranks. In the P2H+ index, nodes are ranked by their degree. In particular, $rank(v)$ of a node v is defined as the rank of the nodes sorted by decreasing order of the degree (where ties are broken arbitrarily). Then, the node w_1 with the largest degree will have $rank(w_1) = 1$, while the node w_n with the smallest degree have $rank(w_n) = n$. The smaller the value $rank(v)$ of a node v is, the higher rank node v has. We further define $rank^{-1}(i)$ as the node with the i-th rank. Notice that the P2H+ index is unique if the rank of the nodes is fixed.

Given vertex ranks, the P2H+ index is then constructed in an iterative manner. In particular, in the i-th iteration, two LC-BFSs are processed from $w_i = rank^{-1}(i)$, i.e., the node with the i-th rank. In particular, a forward (resp. backward) LC-BFS is processed on the graph following the direction (resp. reverse direction) of the edges. We use $L^i_{in}(v)$ and $L^i_{out}(v)$ to indicate the index built in the i-th iteration. Initially, we add an index $\langle w_i, \emptyset, NULL \rangle$ to the in-entry $L^i_{in}(w_i)$ and out-entry $L^i_{out}(w_i)$ of node w_i [1]. Recall that in the LC-BFS, we maintain a set $\mathcal{S}_v$ of label set and edge pairs for each node v. Now, we no longer maintain the set $\mathcal{S}_v$. Instead, we add an index to the in-entry set $L^i_{in}(v)$ (resp. ${}^iL_{out}(v)$) of node v directly. We explain how to add an index to in-entries during the pruned forward LC-BFS. The process to construct the out-entries with the backward LC-BFS works in a similar manner. Given the source w_i, initially for each out-going edge $\langle w_i, v, \lambda_v \rangle$ of w_i, we derive a path P_v via label set $\{\lambda_v\}$ and add each path to Q_1. Next, in each iteration, it de-queues a path P from Q_i where i is the smallest ID such that the queue is non-empty. Assume that the path P is from s to v with a label set Ψ. If P cannot be Ψ-covered by $L^i_{out}(s)$ and $L^i_{in}(v)$, an index $\langle s, \Psi, \cdot \rangle$ is added to the in-entry set $L^i_{in}(v)$. Next, for each out-going edge $\langle v, u, \lambda \rangle$ of v, a path P_u with label set $\Psi_u = \Psi_v \cup \{\lambda\}$ is derived and added to $Q_{|\Psi_u|}$. Otherwise (if P can be Ψ-covered), path P is pruned.

Definition 3 (Skipped path). *A path P from s to v during the LC-BFS is a skipped path if it is covered by the current P2H+ index.*

The pruned forward LC-BFS finishes when the maintained queues are empty. Next, the backward LC-BFS is processed and the out-entries L^i_{out} are constructed. Finally, we assign $L^{i+1}_{in}(v) \leftarrow L^i_{in}(v)$

[1] Notice that adding such an index entry is unnecessary and we include it for the ease of exposition. The implementation does not need to add such index entries.

and $L_{out}^{i+1}(v) \leftarrow L_{out}^{i}(v)$ for each $v \in V$ and then turn to the $(i+1)$-th iteration. The index construction finishes when all n nodes finish the forward and backward LC-BFSs.

The time complexity of the P2H+ index construction algorithm is $O(2^{2|\Lambda|} \cdot n \cdot (n+m))$ as analyzed in [19]. To explain, during pruned LC-BFSs, it needs to examine if it will be covered by the existing P2H+ index for each visited path. This incurs $O(n \cdot 2^{|\Lambda|})$ cost in the worst case as the in-entry and out-entry of a node includes at most $O(n \cdot 2^{|\Lambda|})$ items. Since a forward and backward pruned LC-BFSs visit at most $O(2^{|\Lambda|} \cdot (n+m))$ paths and it needs to conduct $O(n)$ times, the total cost can be bounded by $O(2^{2|\Lambda|} \cdot n^2 \cdot (n+m))$.

Example 2. Still consider the input graph as shown in Figure 1(a). Assume that $rank(i) = i$. To begin, we first add $\langle 1, \emptyset, NULL \rangle$ to $L_{out}^1(1)$ and $L_{in}^1(1)$. Then, we conduct forward LC-BFS from node 1 since it has the highest rank. The initial setting for node 1 is the same as that in Example 1 except that we do not maintain $\mathcal{S}_v$.

Initially, for the out-going edge $\langle 1, 3, a \rangle$ of node 1, a path P_1 is derived and added to queue Q_1. Then, in the first iteration, path P_1 is de-queued from Q_1 and it examines if 1 to 3 via $\{a\}$ can be covered by $L_{out}^1(1)$ and $L_{in}^1(3)$. Since the answer is no, an entry $\langle 1, \{a\}, \cdot \rangle$ is added to $L_{in}^1(3)$. Then, following the outgoing edge of node 3, two paths P_2 from node 1 to node 3 via label set $\{a\}$ and P_3 from 1 to 3 via label set $\{a, b\}$ are derived and added to Q_1 and Q_2, respectively. Next, path P_2 is de-queued and $\langle 1, \{a\}, \cdot \rangle$ is added to $L_{in}^1(4)$. Then, path P_3 is de-queued and can be covered by $L_{out}^1(1)$ and $L_{out}^1(4)$ and thus is pruned. The forward LC-BFS finishes.

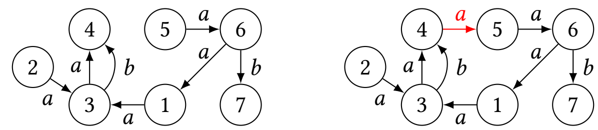
Next, the backward LC-BFS is conducted from node 1. Firstly, a path P_4 from node 1 to 6 via label set $\{a\}$ on the reverse graph is derived and added to queue Q_1. In the first iteration, path P_4 is de-queued. Since the reverse path of P_4 cannot be covered by $L_{out}^1(6)$ and $L_{in}^1(1)$, an entry $\langle 1, \{a\}, \cdot \rangle$ is added to $L_{out}^1(6)$. Then, path P_5 from node 1 to 5 on the reverse graph is derived and added to Q_1. In the next iteration, path P_5 is de-queued. As the reverse path of P_5 cannot be covered by $L_{out}^1(1)$ and $L_{in}^1(5)$, an entry $\langle 1, \{a\}, \cdot \rangle$ is added to $L_{out}^1(5)$. Since node 5 has no incoming edges, the backward LC-BFS finishes. The $L_{in}^1(\cdot)$ and $L_{out}^1(\cdot)$ are then copied as $L_{in}^2(\cdot)$ and $L_{out}^2(\cdot)$, respectively. The index construction finishes and the final index is as shown in Table 2. □

Remark. Due to the interest of space, for the indices we show in Tables 2-4, we omit the entry $\langle v, \emptyset, \cdot \rangle$ from the in-entry and out-entry from each node v; we use a string ab to indicate the set $\{a, b\}$.

Our proposed DLCR is based on the state-of-the-art P2H+ index. Next, we will elaborate on our DLCR for dynamic graphs.

3 DLCR INDEX FOR DYNAMIC GRAPHS

Given an input graph G, assume that after a set of update operations the graph becomes G'. Here we only consider edge insertions/deletions as node insertions/deletions can be easily mapped to a set of edge insertions/deletions. Our main idea is to provide a light-weighted scheme to make full use of the index for G and efficiently update the index, so that the index after the update is the same as that constructed by the P2H+ for G' from scratch. We call this property *update invariant*. Our DLCR is update invariant and it indicates that our DLCR index is unique after each update due to the same vertex order. In addition, the DLCR index is also update-friendly. The key idea is to carefully derive a small set of affected nodes and then the indices are only updated on those affected nodes. We will elaborate on our insertion algorithm in Section 3.1 and the deletion algorithm in Section 3.2.



(a) G_1 (b) G_2

Figure 1: G_1 is the original graph, and G_2 is the updated graph after an edge insertion.

Table 2: Index entries of Figure 1(a).

ID	L_{in}	L_{out}	$InvL_{in}$	$InvL_{out}$
1	-	-	$\langle 3, a, \cdot \rangle$, $\langle 4, a, \cdot \rangle$	$\langle 5, a, \cdot \rangle$, $\langle 6, a, \cdot \rangle$
2	-	-	$\langle 3, a, \cdot \rangle$, $\langle 4, a, \cdot \rangle$	-
3	$\langle 1, a, \cdot \rangle$, $\langle 2, a, \cdot \rangle$	-	$\langle 4, a, \cdot \rangle$, $\langle 4, b, \cdot \rangle$	-
4	$\langle 1, a, \cdot \rangle$, $\langle 2, a, \cdot \rangle$ $\langle 3, a, \cdot \rangle$, $\langle 3, b, \cdot \rangle$	-	-	-
5	-	$\langle 1, a, \cdot \rangle$	$\langle 6, a, \cdot \rangle$, $\langle 7, ab, \cdot \rangle$	-
6	$\langle 5, a, \cdot \rangle$	$\langle 1, a, \cdot \rangle$	$\langle 7, b, \cdot \rangle$	-
7	$\langle 5, ab, \cdot \rangle$, $\langle 6, b, \cdot \rangle$	-	-	-

3.1 Insertion Algorithm

In this section, we present the details of our insertion algorithm. Before introducing the insertion algorithms, we first define an auxiliary data structure to support efficient index updates.

Definition 4 (Inverted 2-hop index). *An inverted 2-hop index is created by inverting the original 2-hop index. Given an index entry $\langle v, \Psi, \cdot \rangle$ in $L_{out}(u)$ (resp. $L_{in}(u)$), an entry $\langle u, \Psi, \cdot \rangle$ exists in $InvL_{out}(v)$ (resp. $InvL_{in}(v)$).*

An example of the inverted index is shown in Table 2. Next, we elaborate on the main idea of our insertion algorithm.

Main idea. With a new edge insertion, new paths will be generated by the existing paths and the added edge, and such paths may not be covered by the current 2-hop index. For example, suppose we add an edge $\langle 4, 5, a \rangle$ in Figure 1(b), then node 1 could reach node 5 via label set $\{a\}$ after this edge is added. However, this path is not covered by existing 2-hop index. So we need to generate new index entries to keep the 2-hop cover property of index entries. For the above new path from node 1 to node 5, we need to add an in-entry $\langle 1, \{a\}, \langle 4, 5, a \rangle \rangle$ into $L_{in}(5)$, which can be generated from the existing in-entry $\langle 1, \{a\}, \cdot \rangle$ in $L_{in}(4)$ and the added edge $\langle 4, 5, a \rangle$. We will elaborate on how to find such new paths shortly. The high-level idea is to apply the pruned LC-BFS to find and add those new entries so that the 2-hop cover property of index entries could be satisfied on the new graph.

After adding those new entries to make the index satisfy the 2-hop cover property, some old entries might become redundant, i.e., deleting them will not violate the 2-hop cover property. We aim to keep the minimum property of the index, and thus such redundant index should be deleted. Table 3 illustrates the index entries after adding the entries to satisfy the 2-hop cover property. The in-entry $\langle 5, \{a\}, \cdot \rangle$ in $L_{in}(6)$ is redundant, since it is 2-hop

Algorithm 1: AddEntries

```
   Input: G′, new edge ⟨u, v, λ⟩, L_in, L_out
   Output: 𝒜_f, 𝒜_b
1  𝒜_f ← ∅, 𝒜_b ← ∅
2  B ← the set of source nodes appear in L_in(u) and L_out(v)
3  for each node x ∈ B in decreasing order of the rank do
4      Let A_I be the set of entries that include x in L_in(u);
5      Initialize Q_0, Q_1, ··· , Q_|λ| to empty set;
6      for each entry ⟨x, Ψ, ·⟩ ∈ A_I do
7          Add entry ⟨v, Ψ ∪ {λ}, ·⟩ to Q_|Ψ∪{λ}|
8      Conduct pruned LC-BFS with Q_0, ··· , Q_|Λ|: add new
         entries during the pruned LC-BFS and add forward
         affected nodes into 𝒜_f.
9      Repeat Lines 4-8 with backward version
10 return 𝒜_f, 𝒜_b
```

covered by $\langle 1, \{a\}, \cdot\rangle$ in $L_{out}(5)$ and $\langle 1, \{a\}, \cdot\rangle$ in $L_{in}(6)$. Thus, the entry can be deleted without violating the 2-hop cover property. By above observations, our insertion algorithm includes two steps:

- *AddEntries.* This phase adds new entries and locates affected nodes (in Definition 5) in forward and backward directions.
- *DeleteRedundant.* It deletes redundant entries on affected nodes.

Next, we explain the rationale behind our design of the two steps.

Rationale and algorithm details. We first consider the property of index entries that are necessary to be added to DLCR (otherwise it will violate the 2-hop cover property). We have the following lemma for such entries.

Lemma 1. *For all entries necessary to be added to DLCR, it covers at least a new path that goes through the inserted edge.*

Proof. Assume that for a newly added index entry, it does not cover any path that goes through the newly added edge. Then, the added index entry together with some other entry covers a new path that does not go across the inserted edge. However, by the property of DLCR, before and after every update, it is update invariant. Thus, it covers any path that does not go through the newly inserted edge. Contradiction. □

Based on Lemma 1, we have Theorem 1 for index entries that are necessary to be added to maintain the 2-hop cover property.

Theorem 1. *For every in-entry (resp. out-entry) that are necessary to be added to DLCR to maintain the 2-hop cover property, it corresponds to a visited path of a forward (resp. backward) pruned LC-BFS from a source node s in $L_{in}(u)$ (resp. $L_{out}(v)$).*

Proof. We use *Query* to check every to-be-added in-entry whether it could cover at least one new path. Once it is added, it indicates this in-entry together with other out-entries could cover at least one new path. Otherwise, this in-entry will not be added. □

Given Theorem 1, the new entries to maintain the 2-hop cover property can be added as follows, where Algorithm 1 shows the pseudo-code for *AddEntries* phase. Firstly, let B be the set of nodes in $L_{in}(u)$ and $L_{out}(v)$ (Algorithm 1 Line 2). Then, following the rank of nodes in B, a forward (resp. backward) LC-BFS is conducted with the node x as the source if x appears in $L_{in}(u)$ (resp. $L_{out}(v)$). Algorithm 1 Lines 4-8 shows the pseudo-code for the forward pruned LC-BFS from node x. Instead of conducting LC-BFS from scratch from x, it only starts from the paths that will go through the newly inserted edge $\langle u, v, \lambda\rangle$. Thus, for each index entry $\langle x, \Psi, \cdot\rangle$, we add an entry $\langle v, \Psi \cup \{\lambda\}, \cdot\rangle$ to $Q_{|\Psi\cup\{\lambda\}|}$ (Algorithm 1 Lines 4-7). Then, a pruned LC-BFS is conducted with the initialized queues $Q_0, \cdots, Q_{|\Lambda|}$. During the pruned LC-BFS, in each iteration, it de-queues a path P from x to a node w via label set Ψ_w with the smallest size of label set. If the path P can be covered by the current 2-hop index, then the path is pruned. Otherwise, the path P is not pruned, then an in-entry $\langle x, \Psi_w, \cdot\rangle$ is inserted to $L_{in}(w)$. Similarly, a backward LC-BFS is conducted if x appears in $L_{out}(v)$ (Algorithm 1 Line 9). According to Theorem 1, after adding such index entries, the 2-hop cover property is satisfied.

Nevertheless, there might exist redundant entries after creating new entries, i.e., deleting such entries will not violate the 2-hop cover property. The key challenge is how to locate these redundant entries efficiently. We first define the affected nodes.

Definition 5 (Affected Nodes). *A set $\mathcal{A}$ of affected nodes is:*

$$\mathcal{A} = \{v \mid L_{in}(v) \text{ or } L_{out}(v) \text{ is changed}\}.$$

If $L_{in}(v)$ (resp. $L_{out}(v)$) has changed, node v is referred to as a forward (resp. backward) affected node in $\mathcal{A}_f$ (resp. in $\mathcal{A}_b$).

The affected nodes help to locate redundant entries as follows.

Theorem 2. *For a forward (resp. backward) affected node $t \in \mathcal{A}_f$ (resp. $\mathcal{A}_b$), if it includes redundant entries, they must exist in $L_{in}(t)$ or $InvL_{out}(t)$ (resp. $L_{out}(t)$ or $InvL_{in}(t)$).*

Proof. We suppose an in-entry $\langle s, \Psi_0, \cdot\rangle$ which in the $L_{in}(t)$ is added and t is the corresponding forward affected node $\mathcal{A}_f$. Consider the $Query(p, q, \Psi)$ process, it tries to locate an out-entry of $L_{out}(p)$ and an in-entry of $L_{in}(p)$ s.t. their id is same and their label set $\subseteq \Psi$. The redundant entries appear because some existing entries might be 2-hop covered by new entries plus existing entries. Thus, with this in-entry insertion, there might be an out-entry, e.g. $\langle s, \Psi_1, \cdot\rangle$ in the $L_{out}(s_0)$, together with the new added in-entry $\langle s, \Psi_0, \cdot\rangle$ in $L_{in}(t)$ s.t. they could 2-hop cover existing entries like $s_0 \overset{\Psi_2}{\rightsquigarrow} t$ where $\Psi_0 \cup \Psi_1 \subseteq \Psi_2$. Since an entry is either in-entry or out-entry, there are two cases for the direction of $s_0 \overset{\Psi_2}{\rightsquigarrow} t$. When $s_0 \overset{\Psi_2}{\rightsquigarrow} t$ is an in-entry, then $\langle s_0, \Psi_2, \cdot\rangle$ will in the $L_{in}(t)$. So we could use $L_{in}(t)$ to find such redundant entries. When $s_0 \overset{\Psi_2}{\rightsquigarrow} t$ is an out-entry, then $\langle t, \Psi_2, \cdot\rangle$ will in the $L_{out}(s_0)$. So we could use $InvL_{out}(t)$ to find such redundant entries. □

According to Theorem 2, we can delete the redundant labels by only inspecting the entries and inverted entries of affected nodes. Here we discuss the case for forward affected nodes, while the case for backward affected nodes can be handled similarly. Algorithm 2 shows the pseudo-code. In particular, for a forward affected node v_f, it tries to delete each entry $e = \langle x, \Psi, \cdot\rangle$ in $L_{in}(v_f)$ and see if the LCR query from v_f to x via label set Ψ can be covered by the 2-hop index without using e (by invoking *QuerySkipEntry*).

Algorithm 2: FwdDelRedundant

Input: v_f, L_{in}, L_{out}
Output: Updated L_{in}, L_{out}
1 **for each** *entry* $\langle x, \Psi, \cdot\rangle \in L_{in}(v_f)$ **do**
2   **if** *QuerySkipEntry*$(x, v_f, \Psi, True)$ **then**
3   ▷ Parameter $True$ indicates query forwardly.
4   Delete this entry from $L_{in}(v_f)$ and $InvL_{in}(x)$
5 **for each** *entry* $\langle x', \Psi', \cdot\rangle \in InvL_{out}(v_f)$ **do**
6   **if** *QuerySkipEntry*$(x', v_f, \Psi', False)$ **then**
7   Delete this entry from $InvL_{out}(v_f)$ and $L_{out}(x')$
8 **return** L_{in}, L_{out}

Algorithm 3: DLCR-edge-insertion

Input: Updated graph G', inserted edge $\langle u, v, \lambda\rangle$, L_{in}, L_{out}
Output: Updated L_{in}, L_{out}
1 **if** *Query*$(u, v, \{\lambda\})$ **then**
2   **return** L_{in}, L_{out}
3 $\mathcal{A}_f, \mathcal{A}_b \leftarrow$ AddEntries$(\langle u, v, \lambda\rangle, G', L_{in}, L_{out})$
4 **for each** *node* $v_f \in \mathcal{A}_f$ **do**
5     $L_{in}, L_{out} \leftarrow$ FwdDelRedundant(v_f, L_{in}, L_{out})
6 **for each** *node* $v_b \in \mathcal{A}_b$ **do**
7     $L_{in}, L_{out} \leftarrow$ BwdDelRedundant(v_b, L_{in}, L_{out})
8 **return** L_{in}, L_{out}

We omit the pseudo-code of *QuerySkipEntry* as it is straightforward. If the answer is yes, entry e is redundant and can be safely removed from $L_{in}(x_f)$ and the corresponding inverted index is removed from $invL_{in}(x)$ (Algorithm 2 Lines 1-4). Next, it examines each entry $e' = \langle x', \Psi', \cdot\rangle$ in $invL_{out}(v_f)$ and checks if deleting the corresponding index $\langle v_f, \Psi', \cdot\rangle$ in $L_{out}(x')$ can still cover the LCR query from x' to v_f via label set Ψ'. If the answer is yes, the entry $\langle v_f, \Psi', \cdot\rangle$ in $L_{out}(x')$ is redundant and is removed from $L_{out}(x')$ and e' is also removed from $invL_{out}(x')$ (Algorithm 2 Lines 5-7). Next, the backward affected nodes are also processed with the *BwdDelRedundant* algorithm. The pseudo-code is omitted as they are handled in a mirror manner.

Algorithm 3 presents the details of the insertion algorithm. Assume that the added edge is $\langle u, v, \lambda\rangle$. It first inspects if u can reach v by λ. If the answer is yes, then the added new edge does not bring any uncovered new paths. The algorithm then immediately terminates (Algorithm 3 Lines 1-2). Otherwise, it indicates that there exist uncovered new paths due to this insertion. To update the index, it adds new entries and finds the affected nodes (Algorithm 3 Line 3). Finally, it deletes redundant index entries by inspecting the index entries and inverted index entries for each affected node (Algorithm 3 Lines 4-7). An example to show how our insertion algorithms works is given as follows.

Example 3. Assume that an edge $\langle 4, 5, a\rangle$ is inserted to G_1 in Figure 1(a) and the updated graph is shown in Figure 1(b). The original index is shown in Table 2. To the beginning, we create new entries using $L_{in}(4)$ and $L_{out}(5)$ which are $\{\langle 1,\{a\},\cdot\rangle, \langle 2,\{a\},\cdot\rangle, \langle 3, \{a\},\cdot\rangle, \langle 3, \{b\},\cdot\rangle\}$ and $\{\langle 1, \{a\}, \cdot\rangle\}$. We start from high-ranking entries to low-ranking entries. In the first iteration, we begin with node 1 and its corresponding in-entry and out-entry is $\{\langle 1, \{a\}, \cdot\rangle\}$ and $\{\langle 1, \{a\}, \cdot\rangle\}$. As for forward version, after applying the pruned LC-BFS, we add $\langle 1, \{a\}, \cdot\rangle$ into $L_{in}(5)$ and $\langle 1, \{a\}, \cdot\rangle$ into $L_{in}(6)$, $\langle 1,\{ab\}, \cdot\rangle$ into $L_{in}(7)$. As for backward version, we add $\langle 1, \{a\}, \cdot\rangle$ into $L_{out}(4)$, $\langle 1, \{a\}, \cdot\rangle$ into $L_{out}(3)$, $\langle 1, \{a\}, \cdot\rangle$ into $L_{out}(2)$. In the second iteration, we begin with node 2 and it has only in-entry $\{\langle 2, \{a\}, \cdot\rangle\}$. Applying the pruned LC-BFS, this entry is pruned by the existing index so this iteration stops. In the third iteration, we begin with node 3 and it has only in-entry $\{\langle 3, \{a\}, \cdot\rangle, \langle 3, \{b\}, \cdot\rangle\}$. Applying the pruned LC-BFS, these two entries are also pruned by existing entries. This iteration terminates and AddEntries phase finishes. The forward affected nodes are 5, 6, and 7 while the backward affected nodes are 2, 3, and 4. Table 3 shows the entries after AddEntries phase. Then it turns to the DeleteRedundant phase. Take backward affected node 6 as an example. We first check entries in $L_{in}(6)$. The label $\langle 1, \{a\}, \cdot\rangle$ is not redundant because *QuerySkipEntry*$(1, 6, \{a\})$ returns $False$ while the entry $\langle 5, \{a\}, \cdot\rangle$ is redundant because *QuerySkipEntry*$(5, 6, \{a\})$ returns $True$. Secondly, we check entries in $InvL_{out}(6)$. Because $InvL_{out}(6)$ is empty, we stop. Table 4 summarizes the final DLCR index. □

Table 3: DLCR index after AddEntries phase. Entries in blue are newly added ones compared with Table 2.

ID	L_{in}	L_{out}	$InvL_{in}$	$InvL_{out}$
1	-	-	$\langle 3, a, \cdot\rangle$, $\langle 4, a, \cdot\rangle$ $\langle 5, a, \cdot\rangle$, $\langle 6, a, \cdot\rangle$ $\langle 7, ab, \cdot\rangle$	$\langle 2, a, \cdot\rangle$, $\langle 3, a, \cdot\rangle$ $\langle 4, a, \cdot\rangle$, $\langle 5, a, \cdot\rangle$ $\langle 6, a, \cdot\rangle$
2	-	$\langle 1, a, \cdot\rangle$	$\langle 3, a, \cdot\rangle$, $\langle 4, a, \cdot\rangle$	-
3	$\langle 1, a, \cdot\rangle$, $\langle 2, a, \cdot\rangle$	$\langle 1, a, \cdot\rangle$	$\langle 4, a, \cdot\rangle$, $\langle 4, b, \cdot\rangle$	-
4	$\langle 1, a, \cdot\rangle$, $\langle 2, a, \cdot\rangle$ $\langle 3, a, \cdot\rangle$, $\langle 3, b, \cdot\rangle$	$\langle 1, a, \cdot\rangle$	-	-
5	$\langle 1, a, \cdot\rangle$	$\langle 1, a, \cdot\rangle$	$\langle 6, a, \cdot\rangle$, $\langle 7, ab, \cdot\rangle$	-
6	$\langle 1, a, \cdot\rangle$, $\langle 5, a, \cdot\rangle$	$\langle 1, a, \cdot\rangle$	$\langle 7, b, \cdot\rangle$	-
7	$\langle 1, ab, \cdot\rangle$, $\langle 5, ab, \cdot\rangle$ $\langle 6, b, \cdot\rangle$	-	-	-

Table 4: DLCR index entries after DeleteRedundant phase.

ID	L_{in}	L_{out}	$InvL_{in}$	$InvL_{out}$
1	-	-	$\langle 3, a, \cdot\rangle$, $\langle 4, a, \cdot\rangle$ $\langle 5, a, \cdot\rangle$, $\langle 6, a, \cdot\rangle$ $\langle 7, ab, \cdot\rangle$	$\langle 2, a, \cdot\rangle$, $\langle 3, a, \cdot\rangle$ $\langle 4, a, \cdot\rangle$, $\langle 5, a, \cdot\rangle$ $\langle 6, a, \cdot\rangle$
2	-	$\langle 1, a, \cdot\rangle$	-	-
3	$\langle 1, a, \cdot\rangle$	$\langle 1, a, \cdot\rangle$	$\langle 4, b, \cdot\rangle$	-
4	$\langle 1, a, \cdot\rangle$, $\langle 3, b, \cdot\rangle$	$\langle 1, a, \cdot\rangle$	-	-
5	$\langle 1, a, \cdot\rangle$	$\langle 1, a, \cdot\rangle$	-	-
6	$\langle 1, a, \cdot\rangle$	$\langle 1, a, \cdot\rangle$	$\langle 7, b, \cdot\rangle$	-
7	$\langle 1, ab, \cdot\rangle$, $\langle 6, b, \cdot\rangle$	-	-	-

An advantage of our insertion algorithm is that the updated index is unique and the same as that built from scratch on G' by the P2H+ index. Next, we show that the DLCR is update invariant and analyze the time complexity of our insertion algorithm.

Correctness and complexity analysis. We next show that the DLCR index is update invariant. Given an edge insertion, let the index after the *addEntries* phase be L' and the index after the *DeleteRedundant* phase be L^*. Let L be the index built from scratch by the

P2H+ index. We first have the following lemma for the index L' after the *AddEnries* phase,

LEMMA 2. *For each node v, $L'_{in}(v)$ (resp. $L'_{out}(v)$) is a superset of $L_{in}(v)$ (resp. $L_{out}(v)$), i.e., $L_{in}(v) \subseteq L'_{in}(v)$ (resp $L_{out}(v) \subseteq L'_{out}(v)$).*

PROOF. Since we conduct the pruned LC-BFS from high rank source nodes to low rank ones, only after the higher rank node has generated new entries for all new paths it could reach, then the lower rank node could start to conduct LC-BFS. And during the entry generation process of high rank node, it will not use the lower rank entries as the pruned condition. It indicates L' is the superset of L because L' has covered all paths using the same vertex order but L' has not deleted redundant entries. □

We have Lemma 3 about the redundancy of an entry.

LEMMA 3. *For each entry in $L_{in}(v)$ (resp. $L_{out}(v)$), it is not redundant in $L'_{in}(v)$ (resp. $L'_{out}(v)$).*

PROOF. Suppose there is an in-entry $\langle s, \Psi, \cdot\rangle$ in $L_{in}(v)$ which is a redundant entry in $L'_{in}(v)$, then it indicates we could find a higher rank node p such that there are an out-entry $s \overset{\Psi_1}{\rightsquigarrow} p$ and an in-entry $p \overset{\Psi_2}{\rightsquigarrow} v$ where $\Psi_1 \cup \Psi_2 \subseteq \Psi$ in L'. Since, this in-entry $\langle s, \Psi, \cdot\rangle$ is not a redundant entry in $L_{in}(v)$ which indicates that either the out-entry $s \overset{\Psi_1}{\rightsquigarrow} p$ or the in-entry $p \overset{\Psi_2}{\rightsquigarrow} v$ does not exist in $L_{in}(v)$. However, since L and L' are constructed with same vertex order, for each node, its index entries must same in L and L'. Contradiction. □

Combining Lemmas 2 and 3, we have the following theorem.

THEOREM 3. *After the insertion algorithm of DLCR, the index L^* is exactly the same as the index L built from scratch by P2H+ index.*

PROOF. According to Lemma 2, we have $L \subseteq L'$. Together with Lemma 3, after deleting the redundant entries in L', then $L = L'$. □

Theorem 3 shows that DLCR is update-invariant after edge insertions. We now consider the time complexity of Algorithm 3.

In the first step, we do pruned LC-BFS from $L_{in}(u)$ and $L_{out}(v)$. Let B be the set of nodes appeared in $L_{in}(u)$ and $L_{out}(v)$. Then, the time complexity for *AddEntries* phase is $O(2^{2^{|\Lambda|}} \cdot |B| \cdot n \cdot (n + m))$. Let L_{add} be the set of added in-entries or out-entries. Then, there are at most $|L_{add}|$ affected nodes. For each affected node, it examines $O(2^{|\Lambda|} \cdot n)$ entries and for each entry, it takes a query and runs with $O(2^{|\Lambda|} \cdot n)$ cost. Thus, the *DeleteRundant* phase takes $O(|L_{add}| 2^{|2\Lambda|} \cdot n^2)$ cost. Adding them together, the time complexity is $O\left(2^{|2\Lambda|} \cdot n \cdot (|L_{add}| \cdot n + |B| \cdot (n + m))\right)$

3.2 Deletion Algorithm

Main idea. With an edge deletion, some paths may disappear such that the current index may return incorrect query answers. We call such index entries outdated entries. For instance, assume that we delete edge $\langle 4, 5, a\rangle$ from G_2 in Figure 1(b) and thus the graph becomes G_1 as shown in Figure 1(a). Then, in-entry $\langle 1, \{a\}, \langle 4, 5, a\rangle\rangle$ in $L_{in}(5)$ is outdated since path $1 \overset{\{a\}}{\rightsquigarrow} 5$ does not exist due to the deletion of edge $\langle 4, 5, a\rangle$. After deleting outdated entries, some skipped

Algorithm 4: FWDDELENTRY

Input: G', $\langle u, v, \lambda\rangle$, L_{in}
Output: $\mathcal{A}_f$

$\mathcal{A}_f \leftarrow \emptyset$, $\mathcal{A}_l \leftarrow \emptyset$
for each *entry* $\langle p, \Psi, e\rangle \in L_{in}(v)$ *and* $e = \langle u, v, \lambda\rangle$ **do**
  $\mathcal{A}_l.push(\langle p, \Psi, e\rangle)$
for each *entry* $\langle s, \Psi, \cdot\rangle \in \mathcal{A}_l$ **do**
  delete $\langle s, \Psi, \cdot\rangle$ from $L_{in}(v)$
  $\mathcal{A}_f.insert(v)$, $Q \leftarrow \{(v, \Psi)\}$
  while $Q \neq \emptyset$ **do**
    $(x, \Psi_1) \leftarrow Q.pop()$
    for each *out-going edge* $\langle x, t, \lambda_1\rangle$ *of* x **do**
      if $rank[t] \leq rank[s]$ **then**
        continue
      $\Psi_x \leftarrow \Psi_1 \cup \{\lambda_1\}$
      if $(s, \Psi_x, \langle x, t, \lambda_1\rangle) \in L_{in}(t)$ **then**
        $Q.push((t, \Psi_x))$
        Delete $\langle s, \Psi_x, \langle x, t, \lambda_1\rangle\rangle$ from $L_{in}(t)$
        $\mathcal{A}_f.insert(t)$
return $\mathcal{A}_f$, L_{in}

paths, which are pruned by the outdated entries before the deletion, need to be reactivated to create new entries. For instance, $\langle 2, 3, \{a\}, \langle 2, 3, a\rangle\rangle$ is a skipped path and needs to be reactivated. To explain, the existing in-entry $\langle 2, \emptyset, \cdot\rangle$ in $L_{in}(2)$ could travel via edge $\langle 2, 3, a\rangle$, and generated new in-entry $\langle 2, \{a\}, \langle 2, 3, a\rangle\rangle$, which is not pruned after deletion and should be inserted into $L_{in}(3)$ (See Table 4). These skipped paths need to be reactivated to add new entries. In summary, the deletion algorithm includes three steps:

- *DeleteEntries.* It deletes outdated entries and finds affected nodes.
- *LocateSkippedPath.* It locates skipped paths to be reactivated.
- *CreateNewEntries.* It conducts pruned LC-BFSs to create new entries using the valid skipped paths found in the second phase.

Rationale and Algorithm details. Next, we explain how the three steps work one by one. In the first step, to find the outdated entries, we have the following lemma.

LEMMA 4. *All outdated entries travel via the deleted edge $\langle u, v, \lambda\rangle$.*

PROOF. The proof is similar to that of Lemma 1. □

Given the deleted edge $\langle u, v, \lambda\rangle$, Lemma 4 demonstrates that all outdated entries travel via this deleted edge. We note that for a given index entry that corresponds to a path P from x to y where x has the highest rank on P, each sub-path of P starting from x corresponds to an index entry as well. The case when y has the highest rank will have the mirror case. To make use of this property, every entry will store a *lastEdge* property, which is the last edge before inserting the corresponding entry during the index construction process via pruned LC-BFS. With this property, we are now able to efficiently locate the index entries that travel via the deleted edge $e = \langle u, v, \lambda\rangle$. In particular, we first examine the index entries in $L_{in}(u)$ and find

Algorithm 5: FwdSkippedPath

Input: Forward affected node t, $\mathcal{P}_f$, $\mathcal{P}_b$, G', L_{in}, $InvL_{out}$
Output: $\mathcal{P}_f$, $\mathcal{P}_b$

1 **for each** *incoming edge* $\langle x, t, \lambda \rangle$ *of* t **do**
2   **for each** *entry* $\langle s, \Psi, \cdot \rangle \in L_{in}(x)$ **do**
3   **if** $Query(s, t, \Psi \cup \{\lambda\})$ **then**
4   continue
5         $\mathcal{P}_f.insert((s, t, \Psi \cup \{\lambda\}, \langle x, t, \lambda \rangle))$
6 **for each** *entry* $\langle y, \Psi, \cdot \rangle \in InvL_{out}(t)$ **do**
7   **for each** *incoming edge* $\langle s, y, \lambda \rangle$ *of* y **do**
8   **if** $Query(s, t, \Psi \cup \{\lambda\})$ **then**
9   continue
10         $\mathcal{P}_b.insert((s, t, \Psi \cup \{\lambda\}, \langle s, y, \lambda \rangle))$
11 **return** $\mathcal{P}_f, \mathcal{P}_b$

out the entries whose *lastEdge* is e. Then, we start from $L_{in}(u)$ and find more outdated entries that travel via edge e.

Algorithm 4 shows the details of how to find the outdated entries according to entries in $L_{in}(v)$ whose *lastEdge* is the deleted edge $\langle u, v, \lambda \rangle$. In particular, it first finds the entries in $L_{in}(v)$ whose *lastEdge* is $\langle u, v, \lambda \rangle$ and adds them to set $\mathcal{A}_l$. (Algorithm 4 Lines 2 -3). Next, for each such entry $\langle s, \Psi, \cdot \rangle$, it first deletes the entry from $L_{in}(v)$ (Algorithm 4 Line 5). Then, it starts a BFS from s except that: *(i)* it initializes the queue with the path P from s to v via label set Ψ (so that it does not need to do the BFS from the source s from scratch) as shown in Algorithm 4 Line 6; *(ii)* During the BFS traversal, if a path has end point t with a rank higher than s, i.e., $rank(t) < rank(s)$, then the path can be pruned as we are checking the paths with s as the highest rank (Algorithm 4 Line 10-11); *(iii)* It only explores the out-going edges $\langle x, t, \lambda_1 \rangle$ such that the index entry corresponding to path $P' = s \overset{\Psi}{\rightsquigarrow} x \xrightarrow{\lambda_1} t$ exists in $L_{in}(t)$ (Algorithm 4 Lines 13-16). For such a path P', it adds the path into the queue and removes the index entry corresponding to P' in $L_{in}(t)$. If the index entry does not exist in $L_{in}(t)$, then the traversal on such a path can terminate since the extended paths from P' will not exist in the index entries either. Similarly, we can do a mirror phase for entries in $L_{out}(u)$. Instead of doing BFS on G', it proceeds a backward BFS following the reverse direction of graph G'. For the interest of space, we omit the pseudo-code for this phase, dubbed as *BwdDelEntry* phase.

After deleting outdated entries, we need to find skipped paths that are pruned by the outdated entries. In such scenarios, we need to reactive such skipped paths and create new entries for such paths if necessary. This is the second step, i.e., *LocateSkippedPath* step, of our deletion algorithm. One of the main challenges in this step is how to find the skipped paths pruned by the outdated entries efficiently. To achieve this, we record the affected nodes, that include outdated entries in the first step, and use the affected nodes to find the skipped paths pruned by the outdated entries.

Theorem 4. *If a path P is a skipped path pruned by a forward affected node $t \in \mathcal{A}_f$ and is not covered by the index after removing outdated entries, then either* (i) *there exists an in-coming edge $\langle x, t, \lambda \rangle$, and an in-entry $\langle s, \Psi, \cdot \rangle \in L_{in}(x)$ together that map to this path P, i.e., $s \overset{\Psi}{\rightsquigarrow} x \xrightarrow{\lambda} t$ is exactly P; or* (ii) *there exists an entry $\langle y, \Psi, \cdot \rangle$ in $InvL_{out}(t)$ and an incoming edge $\langle s, y, \lambda \rangle$ such that $s \xrightarrow{\lambda} y \overset{\Psi}{\rightsquigarrow} t$ corresponds to path P. The case is mirror for backward affected nodes.*

Proof. Assume that an in-entry $\langle s, \Psi_0, \cdot \rangle$ is deleted from the $L_{in}(t)$, and that t is the associated forward affected node. Consider the $Query(p, q, \Psi)$ operation, it always finds an out-entry of $L_{out}(p)$ and an in-entry of $L_{in}(p)$ such that their id is same and their label set is dominated by Ψ. Before deleting this in-entry, this in-entry $s \overset{\Psi_0}{\rightsquigarrow} t$ together with, e.g. an out-entry $s_0 \overset{\Psi_1}{\rightsquigarrow} s$, could 2-hop cover entries like $s_0 \overset{\Psi_2}{\rightsquigarrow} t$ where $\Psi_0 \cup \Psi_1 \subseteq \Psi_2$. Consider how the entry $s_0 \overset{\Psi_2}{\rightsquigarrow} t$ generates. There are two cases. One is that the entry $s_0 \overset{\Psi_2}{\rightsquigarrow} t$ is an in-entry and it is generated by an existing in-entry an edge, e.g., an in-entry $s_0 \overset{\Psi_3}{\rightsquigarrow} t_0$ and an edge $\langle t_0, t, \lambda \rangle$ where $\Psi_3 \cup \lambda = \Psi_2$. Thus, we could find such skipped paths using forward affected node t's In-Neighbor and this In-Neighbor's in-entries. Here, the skipped path is the in-entry $s_0 \overset{\Psi_3}{\rightsquigarrow} t_0$ together with the edge $\langle t_0, t, \lambda \rangle$. Another case is that the entry $s_0 \overset{\Psi_2}{\rightsquigarrow} t$ is an out-entry, and it is generated by an existing out-entry an edge, e.g., an out-entry $s_1 \overset{\Psi_4}{\rightsquigarrow} t$ and an edge $\langle s_0, s_1, \lambda_0 \rangle$ where $\Psi_4 \cup \lambda_0 = \Psi_2$. Hence, we could find such skipped paths using forward affected node t's $InvL_{out}$ and the In-Neighbor of this out-entry's id. The skipped path is the out-entry $s_1 \overset{\Psi_4}{\rightsquigarrow} t$ together with the edge $\langle s_0, s_1, \lambda_0 \rangle$. After deleting the in-entry $s \overset{\Psi_0}{\rightsquigarrow} t$, then these two kinds of skipped paths need to restart to build new index entries s.t. all paths are still 2-hop covered by index entries. □

Theorem 4 indicates that we can use forward (resp. backward) affected nodes to find skipped paths. Algorithm 5 shows the pseudo-code to find skipped paths by a given forward affected node. In particular, for a forward affected node t, it first examines each of its incoming edges to see if there exist any skipped paths due to t. In particular, let $\langle x, t, \lambda \rangle$ be an incoming edge of t, then it goes through each entry $\langle s, \Psi, \cdot \rangle$ in $L_{in}(x)$ and see if the LCR query with s as the source, t as the target via label set $\Psi \cup \{\lambda\}$ can be covered by current index entries. If the answer is yes, then path $P = s \overset{\Psi}{\rightsquigarrow} x \xrightarrow{\lambda} t$ has already been covered. Otherwise, path P is not covered and we add the path to the skipped path (Algorithm 5 Lines 1-5). Then, it turns the second case where a path might be pruned by the forward affected nodes. In particular, it examines each of the entries in $InvL_{out}(t)$. For each entry $\langle y, \Psi, \cdot \rangle$ in $InvL_{out}(t)$, it examines each of the incoming edges of y, and see if the path $s \xrightarrow{\lambda} y \overset{\Psi}{\rightsquigarrow} t$ can be covered by existing index entries. If the answer is yes, then the path can be discarded. Otherwise, the path is added to the skipped path (Algorithm 5 Lines 6-10). This considers the case for the forward affected nodes. Similarly, we can find the skipped paths due to the backward affected nodes. Since they are mirror cases to Algorithm 5, we omit the discussion.

After finding the skipped paths, we use these skipped paths to generate new entries in Algorithm 6. Similar to Algorithm 1, we start from skipped paths with high ranking node to generate new entries. In particular, for the skipped paths that were discovered,

Algorithm 6: GenNewEntries

Input: $\mathcal{P}_f, \mathcal{P}_b, L_{in}, L_{out}, G'$
Output: L_{in}, L_{out}
1 $C \leftarrow \{x | \langle x, \cdot, \cdot, \cdot \rangle \in \mathcal{P}_f$ or $\langle \cdot, x \cdot, \cdot \rangle \in \mathcal{P}_b\}$
2 **for each** *node* $s \in C$ *in decreasing order of the rank* **do**
3 Let $\mathcal{P}_{tmp}$ be the set of paths such that $\langle s, \cdot, \cdot, \cdot \rangle \in \mathcal{P}_f$
4 Initialize $Q_0, Q_1, \cdots, Q_{|\Lambda|}$ to empty set
5 **for each** $(s, t, \Psi, \cdot) \in \mathcal{P}_{tmp}$ **do**
6 Add entry $\langle t, \Psi, \cdot \rangle$ to $Q_{|\Psi|}$
7 Conduct pruned LC-BFS with $Q_0, \cdots, Q_{|\Lambda|}$
8 Repeat Lines 3-7 with $\mathcal{P}_b$
9 **return** L_{in}, L_{out}

we divide them into two sets $\mathcal{P}_f$ and $\mathcal{P}_b$ as shown in Algorithm 5. Set $\mathcal{P}_f$ stores the set of paths that the starting node has the highest rank while set $\mathcal{P}_b$ stores the set of paths where the ending node has the highest rank. For the case when neither of the ending points of a path has the highest rank, then it will not be possible for such a path to be an index entry since otherwise it will be already covered with higher ranked nodes. After dividing such paths into $\mathcal{P}_f$ and $\mathcal{P}_b$. Then, we can conduct forward LC-BFS and backward LC-BFS from the nodes with the highest rank one by one. Let C be the set of nodes with the highest rank in the skipped paths (either the starting node or the ending node) as shown in Algorithm 6 Line 1. Then, for the nodes in decreasing order of the ranks in C, it does pruned LC-BFS in iterations. In the i-th iteration, let the node with the i-th highest rank be s. Then, it retrieves all paths in $\mathcal{P}_f$ such that the starting node of the skipped paths is s. Then, for each such path $\langle s, t, \Psi, \cdot \rangle$, it adds $\langle t, \Psi, \cdot \rangle$ to queue $Q_{|\Psi|}$. Then, it conducts the pruned LC-BFS with the initialized queues (Algorithm 1 Lines 3-7). Next, it retrieves all paths whose ending points are s in P_b and processes a backward LC-BFS on G' and the iteration finishes. When all nodes in B are processed, the new entries are all added and the algorithm terminates.

Algorithm 7 shows the pseudo-code for the deletion algorithm of DLCR. The pseudo-code is self-explanatory. Lines 1-2 show the *DeleteEntries* step. Lines 3-6 show the *LocateSkippedPath* step. Line 7 shows the pseudo-code of *CreateNewEntries* step.

Then, we demonstrate an example of the deletion algorithm.

Example 4. *In Figure 1(a), suppose we have inserted the edge $\langle 4, 5, a \rangle$ which is shown in Table 4 and now we delete this same edge $\langle 4, 5, a \rangle$ to check whether we can get the same index entries as reconstruction given in the Table 2. To begin, we remove outdated entries using $L_{in}(5)$ and $L_{out}(4)$. ForDelEntry firstly finds $\langle 1, \{a\}, \cdot \rangle$ from $L_{in}(5)$ travels through the deleted edge, then this entry is deleted. Then ForDelEntry will find those entries generated by $\langle 1, \{a\}, \cdot \rangle$ from $L_{in}(5)$. In summary, ForDelEntry will remove $\langle 1, \{a\}, \cdot \rangle$ from $L_{in}(5)$, $\langle 1, \{a\}, \cdot \rangle$ from $L_{in}(6)$ and $\langle 1, \{ab\}, \cdot \rangle$ from $L_{in}(7)$. Thus, the forward affected nodes are 5, 6, and 7. BackDelEntry will delete $\langle 1, \{a\}, \cdot \rangle$ from $L_{out}(4)$, $\langle 1, \{a\}, \cdot \rangle$ from $L_{out}(3)$ and $\langle 1, \{a\}, \cdot \rangle$ from $L_{out}(2)$. Then, the backward affected nodes are 2, 3, and 4. Table 5 displays the entries after the elimination of outdated entries. Second, we determine forward and backward skipped paths by examining the forward and backward affected nodes.Finally, we utilize ForBFSWithInit and BackBFSWithInit to create new entries from high rank skipped paths to low rank skipped paths using these skipped paths. We start from the skipped path $(2, 2, \emptyset, e(2, 3, a))$ and generate new entry $\langle 2, \{a\}, e(2, 3, a) \rangle$ into $L_{in}(3)$ and new entry $\langle 2, \{a\}, e(3, 4, a) \rangle$ into $L_{in}(4)$. Then for the skipped path $(3, 3, \emptyset, e(3, 4, a))$, we generate new entry $\langle 3, \{a\}, e(3, 4, a) \rangle$ into $L_{in}(4)$. Finally, using the skipped path $(5, 5, \emptyset, e(5, 6, a))$, we insert new entries $\langle 5, \{a\}, e(5, 6, a) \rangle$ into $L_{in}(6)$ and $\langle 5, \{ab\}, e(6, 7, b) \rangle$ into $L_{in}(7)$. Table 6 summarizes the final index outcome.*

Table 5: Index entries after edge deletion's Deletion.

ID	L_{in}	L_{out}	$InvL_{in}$	$InvL_{out}$
1	-	-	$\langle 3, a, \cdot \rangle$, $\langle 4, a, \cdot \rangle$	$\langle 5, a, \cdot \rangle$, $\langle 6, a, \cdot \rangle$
2	-	-	-	-
3	$\langle 1, a, \cdot \rangle$	-	$\langle 4, b, \cdot \rangle$	-
4	$\langle 1, a, \cdot \rangle$, $\langle 3, b, \cdot \rangle$	-	-	-
5	-	$\langle 1, a, \cdot \rangle$	-	-
6	-	$\langle 1, a, \cdot \rangle$	$\langle 7, b, \cdot \rangle$	-
7	$\langle 6, b, \cdot \rangle$	-	-	-

Table 6: Index entries after edge deletion's Creation.

ID	L_{in}	L_{out}	$InvL_{in}$	$InvL_{out}$
1	-	-	$\langle 3, a, \cdot \rangle$, $\langle 4, a, \cdot \rangle$	$\langle 5, a, \cdot \rangle$, $\langle 6, a, \cdot \rangle$
2	-	-	$\langle 3, a, \cdot \rangle$, $\langle 4, a, \cdot \rangle$	-
3	$\langle 1, a, \cdot \rangle$, $\langle 2, a, \cdot \rangle$	-	$\langle 4, a, \cdot \rangle$, $\langle 4, b, \cdot \rangle$	-
4	$\langle 1, a, \cdot \rangle$, $\langle 2, a, \cdot \rangle$, $\langle 3, a, \cdot \rangle$, $\langle 3, b, \cdot \rangle$	-	-	-
5	-	$\langle 1, a, \cdot \rangle$	$\langle 6, a, \cdot \rangle$, $\langle 7, ab, \cdot \rangle$	-
6	$\langle 5, a, \cdot \rangle$	$\langle 1, a, \cdot \rangle$	$\langle 7, b, \cdot \rangle$	-
7	$\langle 5, ab, \cdot \rangle$, $\langle 6, b, \cdot \rangle$	-	-	-

Correctness and complexity analysis. Finally, we show that the DLCR deletion algorithm is still update-invariant. Let L' be the index after the *DeleteEntries* step and L^* be the index after all three steps. Let L be the index built from scratch by P2H+ index. Firstly, we have the following lemma for index L' after *DeleteEntries* step.

Lemma 5. *After the DeleteEntries step of DLCR deletion algorithm, for each node v, $L'_{in}(v) \subseteq L_{in}(v)$ and $L'_{out}(v) \subseteq L_{out}(v)$.*

Proof. After the *DeleteEntries* step, the remaining entries will not deleted in the further *CreateNewEntries* step. It is because the remaining entries are generated following the minimal property and the vertex order during the index construction. As long as their corresponding paths do not disappear, these entries will not be deleted. As a result, $L' \subseteq L$. □

Next, we have the following theorem for L^*, the index after our DLCR deletion algorithm.

Theorem 5. *The index L^* after the DLCR deletion algorithm is exactly the same as L built from scratch by P2H+.*

Proof. Because we conduct the pruned LC-BFS from high rank skipped paths to low rank ones, then $L^* = L$ due to the same vertex order and minimal property. □

Theorem 5 shows that DLCR deletion is also update-invariant. Next, we analyze the time complexity of DLCR deletion algorithm. Let

Algorithm 7: DLCR-EDGE-DELETION

Input: Updated graph G', deleted edge $\langle u, v, \lambda \rangle$, L_{in}, L_{out}
Output: L_{in}, L_{out}

$\mathcal{A}_f, L_{in} \leftarrow$ FwdDelEntry$(G', \langle u, v, \lambda \rangle, L_{in})$, $\mathcal{P}_f \leftarrow \emptyset$
$\mathcal{A}_b, L_{out} \leftarrow$ BwdDelEntry$(G', \langle u, v, \lambda \rangle, L_{out})$, $\mathcal{P}_b \leftarrow \emptyset$
for each *node* $t \in \mathcal{A}_f$ **do**
  $\mathcal{P}_f, \mathcal{P}_b \leftarrow$ FwdSkippedPath$(t, \mathcal{P}_f, \mathcal{P}_b, L_{in}, InvL_{out})$
for each *node* $t \in \mathcal{A}_b$ **do**
  $\mathcal{P}_f, \mathcal{P}_b \leftarrow$ BwdSkippedPath$(t, \mathcal{P}_f, \mathcal{P}_b, L_{out}, InvL_{in})$
$L_{in}, L_{out} \leftarrow$ GenNewentries$(\mathcal{P}_f, \mathcal{P}_b)$
return L_{in}, L_{out}

B be the set of nodes appeared in $L_{in}(v)$ and $L_{out}(u)$. Then, it conducts pruned BFS (note here it is BFS not LC-BFS) in *DeleteEntries* step from such nodes, whose cost is $O(|B| \cdot (|\Lambda| + \log n)(n + m))$. Then, in the second step, for each affected node, it joins the edges and its in-entries, whose cost can be bounded by $O(n \cdot 2^{|\Lambda|})$. Thus, the cost for the second step is $O(|B| \cdot n \cdot 2^{|\Lambda|})$. Assume that the set C includes the set of node with the highest ranking node in each skipped path. Then, the third step *GenNewEntries* has a cost of $O\left(2^{2^{|\Lambda|}} \cdot |C| \cdot n \cdot (n + m)\right)$, which dominates the cost for the second step. Th final cost is $O\left(\left(|B| \cdot (|\Lambda| + \log n) + |C| \cdot 2^{2^{|\Lambda|}} \cdot n\right)(n + m)\right)$.

4 OPTIMIZATIONS

4.1 Query-Friendly Design

As the input graph is dynamically changing and thus the index dynamically changes, it is natural to adopt an update-friendly index design. Indeed, our initial choice of a data structure for the index structure is to maintain a balanced binary search tree, e.g., RB-tree, so that it gains a good trade-off between index queries and index updates. However, if we carefully analyze the insertion, deletion, and even the index construction algorithms, we can observe that querying with the DLCR index is one of the major subroutines frequently invoked. In particular, during insertion, given an inserted edge $\langle u, v, \lambda \rangle$, the algorithm needs to do forward (resp. backward) pruned LC-BFS from nodes in $L_{in}(u)$ and $L_{out}(v)$ in the *AddEntries* phase. During the pruned LC-BFS, it needs to do a query to check if the path can be pruned or not. In the *DeleteRedundant* phase, it again needs to query frequently to check if an index entry is redundant. In the deletion algorithm, it needs to use queries on existing index entries to locate the skipped paths. During the index construction, to check if a path can be pruned, we need to query with the current index. Our choice of the query-friendly design is mainly motivated by the fact that queries are usually more frequent than index updates in all of the index construction, DLCR insertion algorithm, and DLCR deletion algorithms. Thus, even if the underlying data structure is query-friendly while not update friendly, the benefit brought due to the reduced cost in the query processing can still help improve the overall performance.

To make the index more query-friendly, in the design of index structure, we select the sorted array as the underlying data structure for L_{in} and L_{out}. Compared to the binary search tree design, as for $Query(s, t, \Psi)$, such a design can do query processing with a linear scan of $L_{out}(s)$ of the source node s and $L_{in}(t)$ of the target node t. This makes the query more cache-friendly compared to the binary search tree design. Although this makes the update cost become $O(\ell)$ where ℓ is the size of the index. Before the update, it needs to query the index to check if the entry to be added is redundant or not. Thus, the total cost (the query + the update) can be bounded the query cost even if we adopt the query-friendly design. Hence, the time complexity of the index construction, DLCR insertion algorithm, and DLCR deletion algorithm does not change with the query-friendly dynamic array design. Moreover, in practical implementation, we could delay the update of an array by marking those invalid entries and removing them all together later in the deletion of entries. This further reduces the cost of index updates.

As we will show in the experiment, such a query-friendly design can contribute substantially to query efficiency improvement. With the improved query efficiency, it further helps significantly reduce the index construction cost, insertion cost, and deletion cost. We believe that this observation also sheds light on the design of update algorithms on the 2-hop index for other types of queries, e.g., shortest path and reachability queries.

4.2 Batch Updates

Another optimization of our DLCR is to do batch updates. With batch updates, if collective updates can benefit, then our batch update algorithm can help further improve the update efficiency. We only consider cases when insertions or deletions are in the same batch. In cases where updates are mixed with edge insertions and deletions, they can be processed with a batch insertion followed by a batch deletion. Next, we show our batch update algorithms.

Batch Insertion. Recall that in the insertion algorithm, the first phase adds the new entries and the second phase deletes those redundant entries of the affected nodes. When dealing with multiple edge insertions, we can defer the *DeleteRedundant* phase until all the new entries using the added edges are generated. The reason is that these redundant entries do not affect the correctness and if such redundant entries have a negligible impact on the queries, we can delete redundant entries when the *AddEntries* phase is processed for all nodes in the batch.

To determine if the redundant entries will have a negative impact on the query processing, we present a pre-probing based approach to choose whether to remove redundant entries step by step or postpone the redundant entries removal in a batch. Our pre-probing strategy mainly compares the running time of a batched insertion of a small set of edges, 300 in our case, against the single edge-insertion version. We check if the performance of batched insertion, which postpones the *DeleteRedundant* phase for all nodes, will degrade the performance compared to the insertion algorithm to process edge insertions one by one. If the performance degrades, then the collective insertion will not help reduce the DLCR insertion cost. Thus, we process the DLCR insertion algorithm for each edge one by one. Otherwise, collective insertion helps reduce running cost and we will make full use of the batch insertion.

Batch Deletion. In DLCR deletion algorithm, recall that it includes three steps, which firstly delete outdated entries, then find skipped paths, and finally add the entries. If we delete edges one by one and

Table 7: Statistics of Datasets. ($K = 10^3, M = 10^6, B = 10^9$).

Name	Dataset	V	E	ζ	Synthetic Labels
BG	Biogrid	64K	862K	8	
AX	arXiv	34K	421K	8	✓
ND	NotreDame	325K	1.5M	8	✓
CT	citeseer	384K	1.7M	8	✓
WS	webStanford	281K	2.3M	8	✓
WL	WikipediaLinks(sh)	3.9M	106M	8	✓
T3W	twitterWWW	41M	1.4B	8	✓
FS	friendster	68M	2.5B	8	✓
BIO	bio	5M	11.7M	1391	

Table 8: Indexing time (IT) in seconds, and index size (IS) in megabytes. The index size equals the product of the number of index entries and each entry's size. In this table, "-" indicates that this method is out of memory on this dataset.

Name	DLCR		DLCR-BST		P2H+	
	IT	IS	IT	IS	IT	IS
BG	0.544	8.33	0.69	8.33	2.91	2.08
AX	101	491	1.3K	491	415	123
ND	6.09	124	12.2	124	25	31.1
CT	209	1.56K	1.98K	1.56K	481	391
WS	19.5	234	47.9	234	86	58.9
WL	373	1.17K	1.31K	1.17K	2.39K	292
T3W	3.93K	12.6K	7.49K	12.6K	52.3K	3.16K
FS	6.51K	15.7K	-	-	71.3K	3.92K
BIO	117	1.52K	250	1.52K	73.8	264

each time we process a single-edge in the DLCR deletion algorithm, for the skipped paths discovered, it might become invalid if later an existing edge on the skipped path is deleted. In addition, for the entries added earlier, it might become invalid if an edge on the corresponding path is deleted.

Motivated by this observation, our batch deletion algorithm first deletes the outdated entries for each node in a batch. Then, the set $\mathcal{P}_f$ of forward affected node and $\mathcal{P}_b$ of backward affected nodes are returned. Notice that the set of affected nodes may also overlap in different deletions and this avoids repetitive tasks in the $FwdSkippedPath$ step as well. Then, during the skipped path discovery, it avoids the skipped paths that include any of the deleted edges in the batch, and the size of remaining valid skipped paths will be much smaller compared with single edge deletion. making the second stage *LocateSkippedPath* be much faster. Finally, in the $CreateNewEntries$ step, it again avoids unnecessary computations. To explain, if entries are updated after each edge deletion, the entries generated in $CreateNewEntries$ step by prior deletion might be deleted due to subsequent edge deletions. Our batch deletion separates the outdated entry deletion process from the new entry generation process such that new entries will not be deleted once added into the index. Moreover, in the $CreateNewEntries$ step, it combines multiple skipped paths all together with the same source and can further help reduce the running cost of the pruned LC-BFS. As we will show in the experiment, batch deletion significantly redces the update cost.

Table 9: Query time in nanoseconds. (TQ: true query, FQ: False Query, "-": method out of memory on this dataset.

Name	DLCR		DLCR-BST		P2H+	
	TQ	FQ	TQ	FQ	TQ	FQ
BG	76	79	162	192	752	664
AX	491	1331	2164	8324	3265	3252
ND	140	169	306	421	1484	1411
CT	1331	877	4358	4035	4753	2186
WS	416	276	1255	917	2194	2103
WL	328	207	804	718	3214	2805
T3W	1402	1638	3722	5115	6875	6224
FS	2204	1723	-	-	9953	7219
BIO	4008	813	5255	2053	45427	53646

4.3 Dealing with Large-Label Graphs

To deal with graphs with a large number of labels, P2H+ introduces a two-level index scheme which includes a primary index and a secondary index. The primary index includes a small set Λ' of frequent labels chosen from the label set Λ. Then, the index construction is processed on the subgraph which only includes edges whose label is from Λ'. In P2H+, the size of Λ' is set to 6 and it chooses the 6 labels with the highest frequency. For the secondary index, it first maps all labels in $\Lambda \setminus \Lambda'$ to $|\Lambda'|$ number of virtual labels. The mapping of the labels used in the primary index is consistent with the mapping in the secondary index. In P2H+, the number of virtual labels is thus also 6. Then, the index is constructed on the virtual labels. Given a query from s to t via label set Ψ, it first queries with primary index and sees if s can reach t via $\Psi \cap \Lambda'$. If the answer is yes, then it returns true immediately. Otherwise, it queries with the secondary index. If the secondary index returns false, then the query can return false. Otherwise, it conducts an LC-BFS online to get the query answer. In our update algorithm, we will update the primary index and secondary index using exactly the same DLCR insertion and deletion algorithms.

5 EXPERIMENTS

5.1 Experimental Setup

Setting. We implement all our algorithms in C++ and compile with g++ with full optimization. The source code of P2H+ is kindly provided by its inventors [19]. All experiments are conducted on a Linux machine with Intel Xeon 2.3GHz CPU and 384GB memory.

Datasets & Query sets. We conduct experiments on 9 real-world graph datasets from various types of complex large networks, including social networks, web networks, citation networks, and biological networks. Table 7 summarizes the statistics of tested datasets. All the datasets used in this experiment are publicly available from SNAP[17] and KONECT[16]. For datasets without edge labels, we synthesize edge labels in exponential distribution following the setting in [19]. The number of synthetic labels is set to 8 by default. We compare two kinds of queries which are true queries and false queries. The generation strategy is the same as [19], and we generate each type of query with three types of label set numbers which are 2, 4, and 6. We generate $10,000$ queries for each label set number and calculate the average query time.

Compared Algorithms. We compare the following algorithms.

- **DLCR**[2]**.** The Dynamic algorithm where the underlying data structure is the query-friendly dynamic array for LCR queries, which is introduced in Section 4.1.
- **DLCR-BST.** The Dynamic algorithm whose index entries for the LCR queries are implemented by binary search tree structure.
- **P2H+.** The state-of-the-art 2-hop index for LCR queries [19] on static graphs.

5.2 Experimental Comparisons

Exp 1: Indexing Cost. As shown in Table 8, the index construction time of DLCR is about one order of magnitude smaller than that of P2H+ and is nearly 2x smaller than that of DLCR-BST on average. This is mainly due to the query-friendly design of our DLCR index that reduces the query cost as we will show in Table 9. DLCR-BST is further 2x faster in index construction compared to P2H+ on almost all datasets. The main reason is that DLCR-BST only maintains a single BST for each $L_{in}(v)$ or $L_{out}(v)$ while in P2H+, multiple BSTs are maintained for different nodes in $L_{in}(v)$ or $L_{out}(v)$.

Next, we examine the index size of three methods. Due to the inverted indices and other extra information we maintained in each index entry, e.g., the last edge, the index size of DLCR, and DLCR-BST are about 4x that of P2H+[3]. Then we also conduct experiments on the large label graph, i.e., BIO, which includes more than 1K labels. Following P2H+, the number of virtual labels (Ref. to Section 4.3 for the definition) is set to 6 for large label graphs. Compare with P2H+, DLCR and DLCR-BST take a longer time to build the index. To explain, P2H+ only constructs the secondary index while our DLCR and DLCR-BST construct both the primary and secondary indices to accelerate the query processing.

Exp 2: Query Performance. Table 9 shows that DLCR is nearly one order of magnitude faster than P2H+ in terms of query processing due to the query-friendly design of DLCR. DLCR-BST is still around 2x faster than P2H+. This is consistent with the improvement over the index construction cost.

Exp 3: Deletion Evaluation. In this set of experiments, the deleted edges are chosen randomly from the uniform distribution. The number of deleted edges is set as $10K$, $20K$, $40K$, and $80K$. For each number of deleted edges, we repeat 10 rounds and calculate the average deletion time. Figure 2(a) shows the index update time for edge deletions. Remarkably, our update latency can be as small as a few milliseconds (about 0.1 ms) even on billion edge graphs like T3W and FS. In addition, our batch deletion algorithm significantly reduces the update cost compared to the single-edge DLCR deletion algorithm. In particular, the batch deletion algorithm is up to 2 orders of magnitude faster than the single-edge deletion algorithm, e.g., on AX and CT datasets. To explain, such graphs are citation networks and are well connected, where a single deletion may need to update a large portion of indices. Then, in such scenarios, our batch deletion algorithm thus avoids a large number of unnecessary updates and make batch deletion super efficient on such networks. Moreover, With our batch deletion algorithm, the average update cost is always bounded by no more than 20 milliseconds.

Besides, let m' be the number of edge deletions that we can support using time T with the current index update cost, where T is the index construction time. We report the ratio m'/m, dubbed as the percentage equals to reconstruction, as shown in Figure 2(b). It demonstrates that with our batch deletion index update algorithm, it can update roughly 1% to 10% of edges at once s.t. it is still much faster than reconstruction. In real-world applications, we may not update so many edges such as 1% or 10%. Thus, we only test $10K$, $20K$, $40K$, or $80K$ edges and report the average update performance.

Exp 4: Insertion Evaluation. The settings are similar to that in Exp 3, where the only difference is that we insert the delete edges in Exp 3 back. Figure 3(a) indicates that both single edge insertion and batch insertion methods are efficient. In most datasets, the average insertion time could range between 10^{-3} and 10^{-5} second, while batch insertion could enhance the performance by up to an order of magnitude when collective updates may help, e.g., on citation networks AX and CT. Generally speaking, the DLCR insertion algorithm is around 5x-10x faster than deletion algorithm; the batch insertion algorithm is 2x-5x faster than the batch deletion algorithm.

Figure 2(b) shows the ratio of m'/m, i.e., the percentage equals to reconstruction. The batch insertion algorithm could update nearly 2% to 20% edges at once within the index construction time s.t. it is still faster than reconstruction for most graphs. Moreover, the insertion methods even perform better for billion-scale graphs, e.g., T3W and FS, than the small graphs in terms of the percentage equals to reconstruction.

Exp 5: Large Label Graph Evaluation. From Figure 4, it can be seen that the performance for single and batch insertions is similar, while the batch deletion technique could significantly speed up the performance compared with single deletion algorithm. It is apparent that this trend keeps steady when the number of updated edges increases for both insertion and deletion.

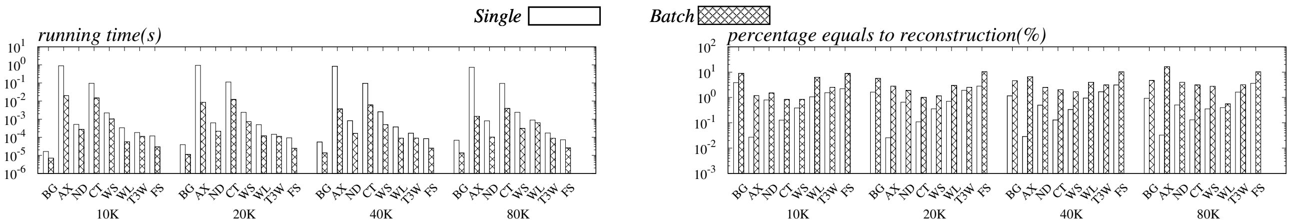
Exp 6: Update Performance of DLCR and DLCR-BST. Exp 6 compares the update performance of DLCR against DLCR-BST. We follow the settings in **Exps 3** and **4**. Figures 5 and 6 show that single (batch) DLCR is nearly 5x faster than single (batch) DLCR-BST in terms of update performance, respectively. To explain, the query-friendly design significantly reduces the update cost.

6 RELATED WORK

Label Constrained Reachability (LCR) Queries. The first work on LCR queries is proposed by [14]. They introduce a new tree-based index framework that utilizes the directed maximal weighted spanning tree algorithm and sampling techniques to condense the generalized transitive closure for labeled graphs to the maximum extent possible. The state-of-art indexing technique is P2H+, a 2-hop index with novel pruning rules and order strategies [19]. Compared with the landmark index-based algorithm[23], P2H+ has smaller index and better query performance.

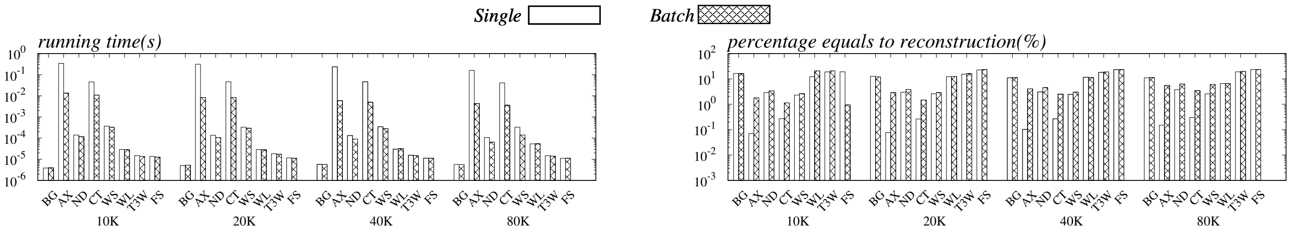
Dynamic Graph Methods. Due to the dynamic nature of real-world networks, there is a pressing need to develop fully dynamic solutions for graph problems. There exist several methods for different types of graph problems. [18] proposes efficient algorithms for hierarchical core maintenance against the insertion/removal of one edge, with effective local update techniques. The algorithms are also extended to handle multiple inserted/removed edges in a

[2]Our code could be found at https://github.com/jerchenxin/DP2H-CUHK
[3]The reported index size equals the product of the number of index entries and the size of each entry in this experiment.



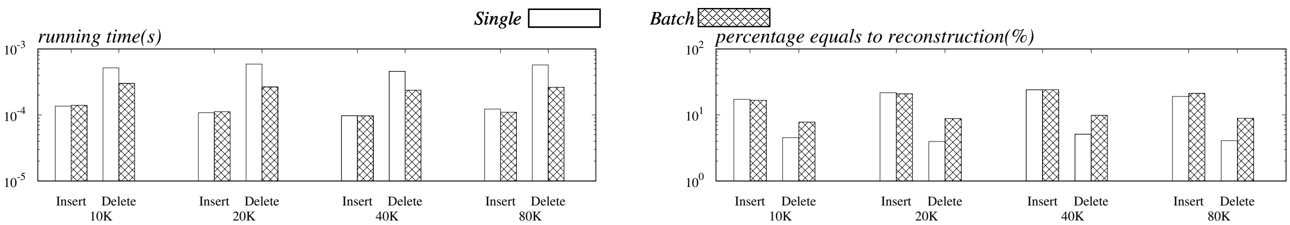
(a) Average deletion time (Lower is better.) (b) Percentage equal to reconstruct (Higher is better.)

Figure 2: Single edge deletion and batch deletion evaluations.



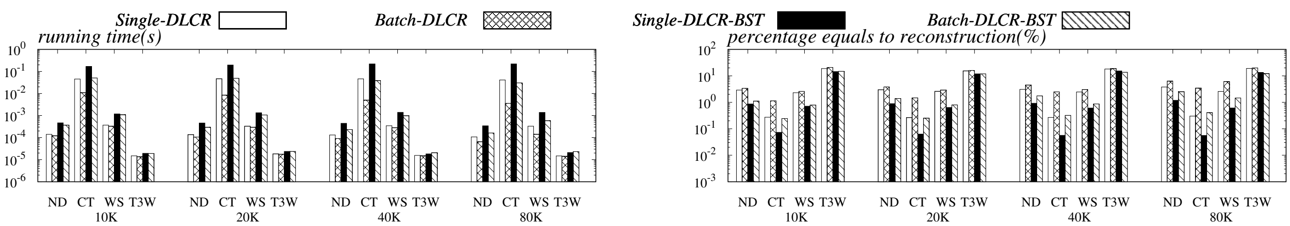
(a) Average insertion time (Lower is better.) (b) Percentage equal to reconstruct (Higher is better.)

Figure 3: Single edge insertion and batch insertion evaluations.



(a) Average insertion and deletion time (Lower is better.) (b) Insertion and deletion: percentage equal to reconstruct (Higher is better.)

Figure 4: Large label graph evaluations with the dataset BIO.



(a) Average insertion time (Lower is better.) (b) Percentage equal to reconstruct (Higher is better.)

Figure 5: Single edge insertion and batch insertion evaluations with DLCR and DLCR-BST.

batch. [9, 11, 20, 21] propose algorithms for incrementally maintaining transitive closures on dynamic graphs. Nevertheless, these methods can not scale to billion-scale graphs which are shown by [15]. [5, 8, 22] propose methods to update the 2-hop labeling index. [13, 27, 28] present algorithms for performing updates on a reachability index. TOL [28] proposes incremental update methods for the 2-hop labeling index, where TOL focuses on removing or inserting a vertex from a graph. However, these methods cannot be extended to LCR queries on dynamic graphs, which is the main contribution of our DLCR.

7 CONCLUSIONS

In this paper, we present DLCR, an efficient 2-hop index based framework for LCR queries on dynamic graphs. Extensive experiments show the efficiency and effectiveness of our proposed algorithms.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*. Springer, 24–35.

[2] 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*. ACM, 349–360.

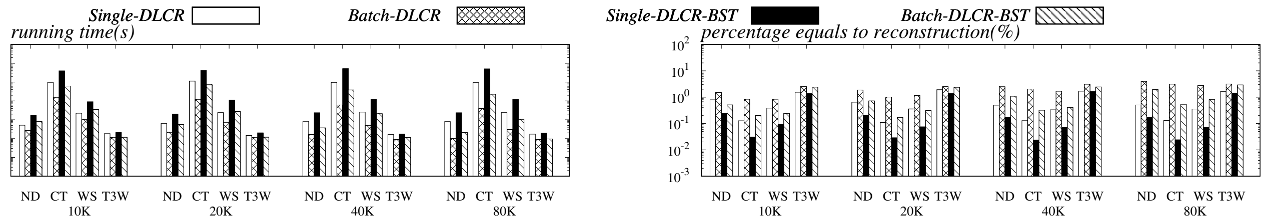
[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.

[4] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.

[5] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. 2010. Incremental Maintenance of 2-Hop Labeling of Large Graphs. *TKDE* 22, 5 (2010), 682–698.

[6] Jiefeng Cheng and Jeffrey Xu Yu. 2009. On-line exact shortest distance query processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, 481–492.

[7] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.



(a) Average deletion time (Lower is better.)

(b) Percentage equal to reconstruct (Higher is better.)

Figure 6: Single edge deletion and batch deletion evaluations with DLCR and DLCR-BST.

[8] Gianlorenzo D'Angelo, Mattia D'Emidio, and Daniele Frigioni. 2019. Fully Dynamic 2-Hop Cover Labeling. *ACM J. Exp. Algorithmics* 24, 1 (2019), 1.6:1–1.6:36.

[9] Camil Demetrescu and Giuseppe F. Italiano. 2006. Fully dynamic all pairs shortest paths with real edge weights. *J. Comput. Syst. Sci.* 72, 5 (2006), 813–837.

[10] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying. In *EDBT*. 520–523.

[11] Monika Rauch Henzinger and Valerie King. 1995. Fully Dynamic Biconnectivity and Transitive Closure. In *FOCS*. 664–672.

[12] 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*. ACM, 123–134.

[13] Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. 2011. Path-tree: An efficient reachability indexing scheme for large directed graphs. *TODS* 36, 1 (2011), 7:1–7:44.

[14] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*. 595–608.

[15] Ioannis Krommidas and Christos D. Zaroliagis. 2008. An experimental study of algorithms for fully dynamic transitive closure. *ACM J. Exp. Algorithmics* 12 (2008), 1.6:1–1.6:22.

[16] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *WWW*. 1343–1350.

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

[18] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. 2021. Hierarchical Core Maintenance on Large Dynamic Graphs. *PVLDB* 14, 5 (2021), 757–770.

[19] 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 (2020), 812–825.

[20] Liam Roditty. 2013. Decremental maintenance of strongly connected components. In *SODA*. 1143–1150.

[21] Liam Roditty and Uri Zwick. 2004. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*. 184–191.

[22] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. 2005. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *ICDE*. 360–371.

[23] Lucien D. J. Valstar, George H. L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *SIGMOD*. 345–358.

[24] 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, CA, USA, June 24 - 24, 2016*. 7.

[25] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently Answering Regular Simple Path Queries on Large Labeled Networks. In *SIGMOD*. 1463–1480.

[26] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Rec.* 41, 1 (2012), 50–60.

[27] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2013. DAGGER: A Scalable Index for Reachability Queries in Large Dynamic Graphs. *CoRR* abs/1301.0977 (2013).

[28] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*. 1323–1334.