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, an important 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 check if there exists any path P from s to t such that labels of edges on P all belong to Ψ. Existing indexing schemes for LCR queries still focus 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 [14, 21, 27], 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, 28] and are supported by practical graph query languages such as SPARQL 1.1, PGQL [26], and openCypher[12]. 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+* [21] 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 [27] 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

Table 1: Frequently used notations

Notations	Definitions
Λ	The set of labels in the input graph
$s \overset{\Psi}{\rightsquigarrow} v$	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 v
$InvL_{in}(v)$, $InvL_{out}(v)$	The inverted in and out 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

exact query results. Our solution is based on the state-of-the-art P2H+ 2-hop index [21]. 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. Although the ideas to find affected nodes for updates are not new, the devil is in the details. It is unclear how to identify these affected nodes efficiently. Given an edge insertion $\langle u, v, \lambda \rangle$, a naive solution is to proceed forward and backward LC-BFS traversals (Ref. to Section 2.3) from scratch from u and v to find the affected nodes. However, such a solution is still computational intensive. To avoid such a drawback, we show that we can actually use existing index entries to quickly re-store the LC-BFS without doing the BFS from scratch. The correctness of such pruning strategies is further verified by our theoretical analysis. After the edge insertion, there may exist redundant edges, which negatively affected the query efficiency of our DLCR scheme. We further present effective solutions by examining label entries without any traversal. This significantly reduces the cost to identify redundant labels. For deletion algorithms, given an edge deletion to $\langle u, v, \lambda \rangle$, similarly, we can re-store the LC-BFS with existing index entries to remove obsoleted entries instead of conducting the LC-BFS from scratch. After the deletion of the obsoleted entries, the index may still return wrong answers as it missed entries that are pruned during the index construction. To re-discover such pruned entries, we show how to again explore the existing index entries to identify the affected nodes and add those entries efficiently. Our theoretical analysis shows that such an efficient solution (via exploring existing index entries) is correct.

Moreover, graph updates may come in batches. We further devise batch update algorithms so that when collective updates can benefit, our batch update algorithm improves the practical efficiency. We show how to devise the batch update algorithms for insertion and deletion and discuss rationales behind the improved efficiency. For our batch insertion (resp. deletion) algorithm, it achieves up to one (resp. two) order(s) of magnitude speedup over insertion (resp. deletion) algorithm that processes updates one by one.

Finally, 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 be the 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 insertion/deletion/query algorithms with the query-friendly design achieve the same time complexity as those with update-friendly data structures.

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.
- *Batch Updates.* Additionally, batch insertion and deletion techniques are proposed to further enhance performance.
- *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.
- *Efficiency and Effectiveness.* Our experiments show the effectiveness and efficiency of our algorithms compared 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 $s \overset{\Psi}{\rightsquigarrow} t$, 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 $s \overset{\Psi}{\not\rightsquigarrow} t$. The label constrained reachability query is defined as follows.

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)* $s \overset{\Psi}{\rightsquigarrow} t$ and *(ii)* $s \overset{\Psi'}{\not\rightsquigarrow} t$ 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, 7, 8]). 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 $s \overset{\Psi}{\rightsquigarrow} t$ 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. First, we 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. [21].

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.

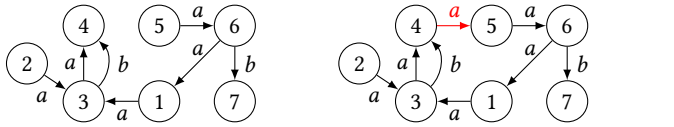
Next, path P_2 from node 1 to 4 via label set $\{a\}$ is de-queued from Q_1. Since S_4 is empty, an entry $\langle \{a\}, \cdot \rangle$ is added to S_4. Since node 4 has no out-neighbors, the iteration finishes. After that, path P_3 from node 1 to 4 via label set $\{a, b\}$ is de-queued from Q_2 since 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



(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.

$L^i_{in}(w_i)$ and out-entry $L^i_{out}(w_i)$ of node w_i [1]. Recall that in LC-BFS, we maintain a set $\mathcal{S}_v$ of label set and edge pairs for each node v. Now, we no longer maintain 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 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), 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 $L^{i+1}_{in}(v) \leftarrow L^i_{in}(v)$ and $L^{i+1}_{out}(v) \leftarrow L^i_{out}(v)$ 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 $O(2^{2|\Lambda|} \cdot n^2 \cdot (n+m))$ as analyzed in [21]. 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 $L^1_{out}(1)$ and $L^1_{in}(1)$. 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 $L^1_{out}(1)$ and $L^1_{in}(3)$. Since the answer is no, an entry $\langle 1, \{a\}, \cdot\rangle$ is added to $L^1_{in}(3)$. 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 $L^1_{in}(4)$. Then, path P_3 is de-queued and can be covered by $L^1_{out}(1)$ and $L^1_{out}(4)$ 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 $L^1_{out}(6)$ and $L^1_{in}(1)$, an entry $\langle 1, \{a\}, \cdot\rangle$ is added to $L^1_{out}(6)$. 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 $L^1_{out}(1)$ and $L^1_{in}(5)$, an entry $\langle 1, \{a\}, \cdot\rangle$ is added to $L^1_{out}(5)$. Since node 5 has no incoming edges, the backward LC-BFS finishes. The $L^1_{in}(\cdot)$ and $L^1_{out}(\cdot)$ are then copied as $L^2_{in}(\cdot)$ and $L^2_{out}(\cdot)$, 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. Our DLCR index is defined as follows:

Definition 4 (DLCR). *Given a $rank(v)$ of each vertex v, the DLCR index is a minimum 2-hop index constructed according to the pruned LC-BFS following the rank of each vertex. Given an edge update, denote the updated graph as G', the updated index satisfies the update invariant property, i.e., the index is the same as that of re-constructing from scratch using the pruned LC-BFS following the rank order of each vertex on G'.*

Our DLCR index is also update-friendly as we will show shortly. Instead of re-constructing from scratch, our DLCR carefully derives a small set of affected nodes and then 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.

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 5 (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.

[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.

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	$\langle 5, ab, \cdot\rangle$, $\langle 6, b, \cdot\rangle$	-	-	-

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*, which says that 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 $\Psi_{out} \subseteq \Psi$. 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 pruned LC-BFSs 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 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 6) 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. □

Algorithm 1: AddEntries

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

1 $\mathcal{A}_f \leftarrow \emptyset$, $\mathcal{A}_b \leftarrow \emptyset$
2 $B \leftarrow$ the set of source nodes appear in $L_{in}(u)$ and $L_{out}(v)$
3 **for each** *node* $x \in B$ *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 $Q_0, Q_1, \cdots, Q_{|\lambda|}$ to empty set;
6 **for each** *entry* $\langle x, \Psi, \cdot\rangle \in A_I$ **do**
7 Add entry $\langle v, \Psi \cup \{\lambda\}, \cdot\rangle$ to $Q_{|\Psi\cup\{\lambda\}|}$
8 Conduct pruned LC-BFS with $Q_0, \cdots, Q_{|\Lambda|}$: add new entries during the pruned LC-BFS and add forward affected nodes into $\mathcal{A}_f$.
9 Repeat Lines 4-8 with backward version
10 **return** $\mathcal{A}_f$, $\mathcal{A}_b$

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.

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** $\mathit{QuerySkipEntry}(x, v_f, \Psi, True)$ **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** $\mathit{QuerySkipEntry}(x', v_f, \Psi', False)$ **then**
7 Delete this entry from $InvL_{out}(v_f)$ and $L_{out}(x')$
8 **return** L_{in}, L_{out}

Definition 6 (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 $\langle s, \Psi_0, \dot{\rangle}$ 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. $\langle s, \Psi_1, \dot{\rangle}$ in the $L_{out}(s_0)$, together with the new added in-entry $\langle s, \Psi_0, \dot{\rangle}$ 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*). 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: 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** $\mathit{Query}(u, v, \{\lambda\})$ **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}

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.

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	$\langle 1, ab, \cdot\rangle$, $\langle 5, ab, \cdot\rangle$ $\langle 6, b, \cdot\rangle$	-	-	-

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

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	$\langle 1, ab, \cdot\rangle$, $\langle 6, b, \cdot\rangle$	-	-	-

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. □

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 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 P2H+. We first have Lemma 2 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 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

Algorithm 4: FwdDelEntry

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

1 $\mathcal{A}_f \leftarrow \emptyset$, $\mathcal{A}_l \leftarrow \emptyset$
2 **for each** *entry* $\langle p, \Psi, e \rangle \in L_{in}(v)$ *and* $e = \langle u, v, \lambda \rangle$ **do**
3 $\mathcal{A}_l.push(\langle p, \Psi, e \rangle)$
4 **for each** *entry* $\langle s, \Psi, \cdot \rangle \in \mathcal{A}_l$ **do**
5 delete $\langle s, \Psi, \cdot \rangle$ from $L_{in}(v)$
6 $\mathcal{A}_f.insert(v)$, $Q \leftarrow \{(v, \Psi)\}$
7 **while** $Q \neq \emptyset$ **do**
8 $(x, \Psi_1) \leftarrow Q.pop()$
9 **for each** *out-going edge* $\langle x, t, \lambda_1 \rangle$ *of* x **do**
10 **if** $rank[t] \leq rank[s]$ **then**
11 **continue**
12 $\Psi_x \leftarrow \Psi_1 \cup \{\lambda_1\}$
13 **if** $(s, \Psi_x, \langle x, t, \lambda_1 \rangle) \in L_{in}(t)$ **then**
14 $Q.push((t, \Psi_x))$
15 Delete $\langle s, \Psi_x, \langle x, t, \lambda_1 \rangle \rangle$ from $L_{in}(t)$
16 $\mathcal{A}_f.insert(t)$
17 **return** $\mathcal{A}_f$, L_{in}

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 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

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** $\mathcal{P}_f$, $\mathcal{P}_b$

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

Algorithm 6: GenNewEntries

Input: $\mathcal{P}_f, \mathcal{P}_b, L_{in}, L_{out}, G'$
Output: L_{in}, L_{out}

1 $C \leftarrow \{x | \langle x, \cdot, \cdot, \cdot \rangle \in \mathcal{P}_f \text{ or } \langle \cdot, x \cdot, \cdot \rangle \in \mathcal{P}_b\}$
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}

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, 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. Due to the interest of space, an example of our deletion algorithms is omitted and could be found in our technical report.

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 (Update Invariant). *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 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. The final cost is $O\left(\left(|B| \cdot (|\Lambda| + \log n) + |C| \cdot 2^{2|\Lambda|} \cdot n\right)(n + m)\right)$.

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}

Remark. It is worth noting that the updated indexes do not have to be the same. Nevertheless, *update invariant* is an elegant state since it is invariant and also convenient to verify the correctness of updated index if such a property is satisfied.

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.

Remark. We note that it is also possible to adopt hashing as a replacement. However, it will not help reduce the search complexity. Recall that given an input source s, a target node t, and a label set Ψ, the LCR query algorithm of DLCR needs to find a middle node v such that node v exists in both $L_{out}(s)$ and $L_{in}(t)$, and v Ψ-covers s to t. Even with hashing, we cannot find the node v in advance. That means, in the worst case, every node v in $L_{out}(s)$ needs to be checked to see if *(i)* the label Ψ-covers s to v, and *(ii)* using hashing to find the labels in $L_{in}(s)$ and go through all labels pertinent to v to check if there exists a label that Ψ-covers v to t. Thus, the time complexity of hashing based solutions is the same as our array design. Moreover, with hashing, it incurs more space consumption and worse cache locality than the array design since array design only needs to scan $L_{out}(s)$ and $L_{in}(t)$ to answer the query.

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 insertion, the first phase adds new entries and the second phase deletes redundant entries of 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

Table 5: Statistics of Datasets. ($K = 10^3, M = 10^6, B = 10^9$).

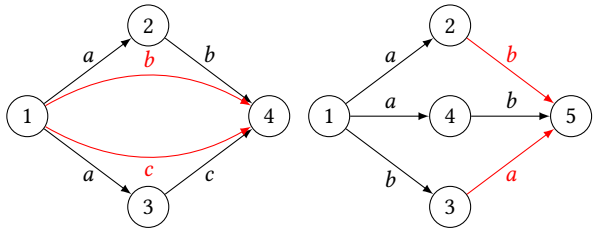
Name	Dataset	V	E	Λ	Synthetic Labels
RB	robots	1.4K	2.9K	4	
AG	advogato	5.4K	51.3K	4	
AX	arXiv	34.5K	422K	8	✓
EP	epinions	132K	841K	8	✓
SH	StringHS	17K	1.24M	7	
ND	NotreDame	326K	1.47M	8	✓
BG	BioGrid	64K	1.58M	7	
CT	citeseer	384K	1.75M	8	✓
SF	StringFC	15.5K	2.04M	7	
WS	webStanford	282K	2.3M	8	✓
WG	webGoogle	876K	5.1M	8	✓
YT	Youtube	15K	10.7M	5	
ZS	zhishihudong	2.45M	18.9M	8	✓
SP	socPokec	1.6M	30.6M	8	✓
WL	wikiLinks	3M	102M	8	✓
DBP	DBPediaLink	18.3M	172M	8	✓
WLE	WikiLinksEng	12.2M	378M	8	✓
T3W	TwitterWWW	41.7M	1.47B	8	✓
TM	TwitterMPI	52.6M	1.96B	8	✓
FS	friendster	68M	2.5B	8	✓
SPL	socPokecLarge	1.6M	30.6M	7513	✓
FB	freebase	14.4M	107M	779	

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 entries. If we delete edges one by one and 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. For the entries added earlier, it might become invalid if an edge on the path is deleted.

Motivated by this observation, our batch deletion algorithm first deletes 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 skipped paths that include any of 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* much faster. Finally, in the *CreateNewEntries* step, it again reduces computations. To explain, if entries are updated after each edge deletion, 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 help reduce the running cost of pruned LC-BFSs. As we will show in the experiment, batch deletion significantly reduces update costs.

We have further included an example to show how the batch insertion and deletion may support more efficient update.



(a) Batch Insertion Example (b) Batch Deletion Example

Figure 2: Batch Insertion and Deletion Examples.

Table 6: DLCR index entries during batch insertion phase.

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

Table 7: DLCR index entries after DeleteRedundant phase.

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

Table 8: DLCR index entries after CreateNewEntries phase.

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

Example 4. *In Figure 2(a), the red edges will be inserted using the batch insertion method. The index entries during the insertion phase is shown in Table 6 where the red entries are the new added entries and the blue entries are the redundant entries. During the batch insertion phase, it will generate new index entries using the new added edges edge by edge. After that, using the affected nodes, it will delete the redundant entries all together. We could see these redundant entries are both in $L_{in}(4)$. Using the batch insertion method, we could only check the index entries in $L_{in}(4)$ once while we need to check twice if we use the single edge insertion method. This is where the batch insertion method could accelerate the insertion process.*

In Figure 2(b), the red edges will be deleted using the batch deletion method. Table 7 shows the index after the first stage DeleteEntries where the blue entries are the entries to be deleted. For each edge, batch deletion method will delete its corresponding outdated entries edge by edge and collect affected nodes. Then we find the pruned path using the affected nodes and the remaining valid index entries. In this simple example, the only pruned path is $\langle 1, 4, a, \langle 4, 5, b\rangle\rangle$. Then using

Table 9: Indexing time or initialization time (IT) in seconds, index size (IS) in megabytes. The index size equals the product of the number of index entries and the size of each entry. ("OOM": out of memory, "OOT": initialization time $\geq 24h$.)

Name	DLCR		DLCR-BST		P2H+		ARRIVAL
	IT	IS	IT	IS	IT	IS	IT
RB	0.00797	0.243	0.00975	0.243	0.011	0.061	0.0222
AG	0.0251	0.926	0.025	0.926	0.069	0.232	0.0465
AX	100	470	1.46K	470	334	118	0.307
EP	1.11	23.4	1.69	23.4	4.4	5.85	0.632
SH	0.52	5.01	0.72	5.01	2.53	1.25	0.583
ND	5.91	138	12.3	138	14.8	34.6	0.969
BG	0.815	13.5	1.24	13.5	4.2	3.38	0.856
CT	208	1.51K	1910	1.51K	374	378	1.79
SF	0.827	4.66	1.1	4.66	3.79	1.17	0.952
WS	18.4	240	50.8	240	49.9	60.1	2.04
WG	69.1	760	163	760	193	190	16.8
YT	140	68	739	68	840	17	5.63
ZS	2.07K	10.7K	17.5K	10.7K	6.34K	2.68K	19.7
SP	79.8	675	142	675	432	169	35.9
WL	158	989	319	989	1.24K	247	87.5
DBP	393	3.83K	672	3.83K	3.43K	956	167
WLE	544	3.3K	1.06K	3.3K	4.87K	824	300
T3W	3.69K	14.6K	7.39K	14.6K	40.6K	3.64K	OOT
TM	4.74K	17.6K	9.94K	17.6K	52.6K	4.4K	OOT
FS	4.37K	19.1K	OOM	OOM	50.6K	4.78K	OOT
SPL	68.2	1.25K	83.6	1.25K	713	313	42.9
FB	1.16K	26.3K	1.59K	26.3K	8.54K	6.58K	5.43K

this pruned path, the third stage CreateNewEntries will generate new index entries which is shown in Table 8 and marked red. Consider we delete these two edges one by one, after the first edge $\langle 2, 5, b\rangle$ is deleted, then the new in-entry $\langle 1, ab, \langle 3, 5, a\rangle\rangle$ will be added into $L_{in}(5)$ which will be deleted when we delete the second edge $\langle 3, 5, a\rangle$. This is why the batch deletion method will accelerate the deletion process.

Time Complexity Assume that there are c updates in a batch. If they are all insertion, then the time complexity of batch insertion is $O\left(2^{|2\Lambda|} \cdot n \cdot (|L_{add}| \cdot n + |B| \cdot (n+m))\right)$, which is the same as the single insertion in Section 3.1). However, notice that the number of affected nodes $|L_{add}|$ in these batch tends to be much larger than that of single insertion. Moreover, notice that in the worst case, $|L_{add} = n|$, and we have the same time complexity as that of index reconstruction. Similarly, the time complexity of batch deletion is $O\left(\left(|B| \cdot (|\Lambda| + \log n) + |C| \cdot 2^{2|\Lambda|} \cdot n\right)(n+m)\right)$, which is the same as that of single deletion in Section 3.2).

4.3 Dealing with Large-Label Graphs

To be self-contained, we briefly recall how P2H+ tackle large labels and then illustrate how to integrate into DLCR and how to dynamically update the indices. 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.

Table 10: Query time in nanoseconds. (TQ: true query, FQ: false query, Acc: Accuracy. "OOM" & "OOT": Ref. to Table 9.

Dataset	DLCR		DLCR-BST		P2H+		ARRIVAL			
	TQ	FQ	TQ	FQ	TQ	FQ	TQ	Acc.	FQ	Acc.
RB	21	1.56K	27	7.51K	296	204	65.1K	0.582	9.57K	1
AG	22	478	39	47.2K	268	237	210K	0.973	20.2K	1
AX	270	4.82K	948	70.3K	2.53K	2.53K	198K	0.0078	72K	1
EP	67	2.39K	127	11.6K	911	786	591K	0.98	38.2K	1
SH	44.5	2.51K	104	13K	484	396	253K	0.985	52.1K	1
ND	97	1.19K	216	8.43K	1.37K	1.33K	785K	0.104	268K	1
BG	75	885	95	10K	755	676	1.48M	0.562	1.82M	1
CT	1.04K	3.76K	4.27K	24.6K	4.48K	2.09K	699K	0.0085	94.5K	1
SF	41.5	779	68.5	1.21K	480	401	263K	0.984	97.9K	1
WS	349	1.79K	1.06K	3.73K	2.06K	1.83K	420K	0.0337	209K	1
WG	370	1.63K	1.14K	5.12K	2.49K	1.92K	3.25M	0.0203	305K	1
YT	477	1.48K	1.89K	5.46K	4.21K	4.37K	1.42M	0.427	1.33M	1
ZS	685	3.76K	2.35K	13K	3.83K	2.88K	1.87M	0.01	157K	1
SP	120	1.31K	176	2.12K	2.25K	2.13K	6.99M	0.985	88K	1
WL	130.5	1.27K	200	2.06K	2.69K	2.49K	8.86M	0.966	114K	1
DBP	163	1.37K	207	2K	4.16K	3.75K	13.1M	0.908	72.5K	1
WLE	169	1.4K	196	1.99K	4.01K	3.61K	11.5M	0.314	91.4K	1
T3W	190	1.69K	220	2.53K	5.5K	5.12K	OOT	OOT	OOT	OOT
TM	196	1.75K	263	17K	5.74K	5.34K	OOT	OOT	OOT	OOT
FS	229	4.84K	OOM	OOM	5.83K	4.89K	OOT	OOT	OOT	OOT
SPL	629K	3.72M	633K	3.76M	7.57B	2.82B	2.1M	0.938	43.3K	1
FB	31.1M	43M	30.9M	42.7M	531M	54.2M	1.39M	0.845	374K	1

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 [21]. All experiments are conducted on a Linux machine with Intel Xeon 2.3GHz CPU and 384GB memory.

Datasets & Query sets. We conduct experiments on 22 real-world graph datasets from various types of complex large networks, including social networks, web networks, citation networks, and biological networks. Table 5 summarizes the statistics of tested datasets. All the datasets used in this experiment are publicly available from SNAP[19] and KONECT[18]. For datasets without edge labels, we synthesize edge labels in exponential distribution following the setting in [21]. The number of synthetic labels is set to 8 by default. The number $|\Lambda'|$ for the large label graphs is set to 4 by default. We compare two kinds of queries which are true queries and false queries. The generation strategy is the same as [21], 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 our DLCR, which adopts the query-friendly dynamic array design, against the following methods: *DLCR-BST*, which uses binary search tree to stores the index; *ARRIVAL*, the state-of-the-art approximate algorithm for

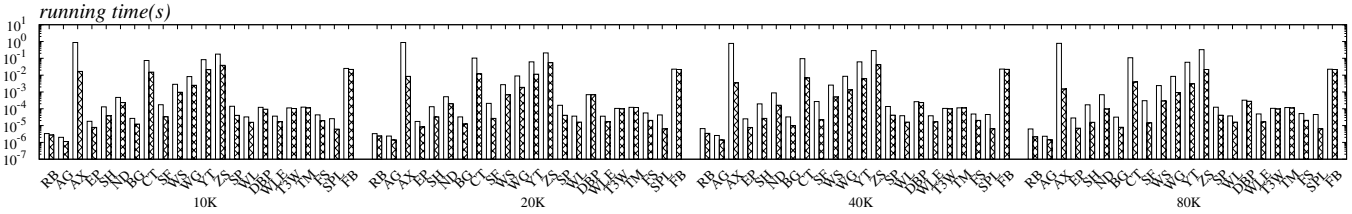


Figure 3: Single edge deletion and batch deletion evaluations.

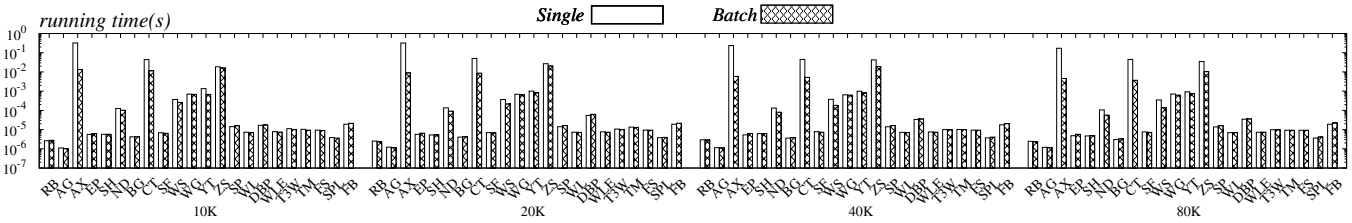


Figure 4: Single edge insertion and batch insertion evaluations.

regular path queries [27] on dynamic graphs; *P2H+*, the state-of-the-art 2-hop index for LCR queries [21] on static graphs.

5.2 Experimental Comparisons

Exp 1: Indexing Cost. As shown in Table 9, 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 10. 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+. Additionally, the initialization time of ARRIVAl can not be ignored as its initialization for T3W, TM and FS cannot finish within 24 hours while our DLCR finishes the index construction for all 22 datasets within one and a half hours.

Exp 2: Query Performance. For the true query set, Table 10 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+. DLCR is nearly four orders of magnitude faster than ARRIVAL while DLCR could give the exact answer. In contrast, ARRIVAL may return approximate solutions with low accuracy on both real labeled graph (e.g., RB with 58%, BG with 56%, YT with 43%, and FB with 85%) and synthetically-generated labeled graphs (e.g., ND with 10%, WG with 2%, and WLE with 32%) on true queries. Table 10 also shows that DLCR is nearly 4x faster than P2H+ for the false query set. DLCR-BST will be around 2x faster than P2H+ in most datasets. In addition, DLCR is nearly two order of magnitude faster than ARRIVAL. For false queries, notice that ARRIVAL will always achieve 100% accuracy.

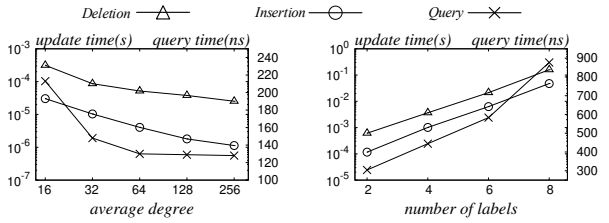
Exp 3: Deletion Evaluation. In this set of experiments, the deleted edges are chosen randomly from the uniform distribution. The number of deleted edges is set as $10K$, $20K$, $40K$, and $80K$. For each number of deleted edges, we repeat 10 rounds and calculate the average deletion time. Remarkably, our update latency can be as small as a few milliseconds (about 0.1 ms) even on billion edge graphs like T3W, TM 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. 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 tens of milliseconds.

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 4 back. Figure 4 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.

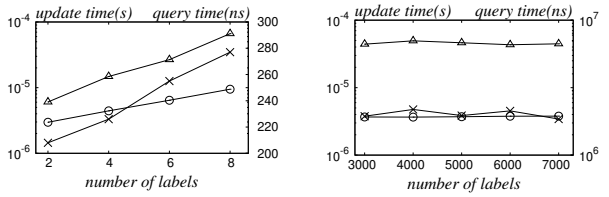
Exp 5: Realistic Query Workload Evaluation. we generate queries that reflect more practical workload in real applications following [5]. To be more precise, we follow the setting in [5] and generate three kinds of query workloads, each containing 100 chain, cycle and star queries, respectively. Each workload is generated by different lengths varying from 4 to 7. The details are shown in Table 11. Notice that it is quite time consuming to find such patterns to generate the query workload. There might not exist certain patterns on such graphs, e.g., cycles, after an exhaustive search with 24 hours. Thus, we focus on 4 datasets: BG, SF, WLE, and FS, where all workloads can be generated efficiently. The results are shown

Table 11: SPARQL workload average query time in ns ($10^{-9}s$). ("OOT": initialization time $\geq$ 24 hours, "OOM": out of memory)

Query Type	Dataset	Length=4				Length=5				Length=6				Length=7			
		DLCR	DCLR-BST	P2H+	ARRIVAL	DLCR	DCLR-BST	P2H+	ARRIVAL	DLCR	DCLR-BST	P2H+	ARRIVAL	DLCR	DCLR-BST	P2H+	ARRIVAL
chain	BG	839	1.24K	3.63K	74.7M	955	1.44K	4.21K	81.8M	1.24K	1.73K	4.93K	98.3M	992	1.37K	5.09K	120M
	SF	447	724	1.95K	22.7M	541	857	2.23K	29.6M	579	913	2.69K	31.2M	714	1.18K	3.1K	34.7M
	WLE	1.25K	2.6K	17.7K	3.02B	1.38K	3.12K	23.3K	3.83B	1.64K	3.77K	24.4K	4.89B	1.86K	4.29K	28.6K	5.08B
	FS	1.43K	OOM	19.6K	OOT	1.37K	OOM	21.5K	OOT	1.55K	OOM	24.8K	OOT	1.69K	OOM	23.1K	OOT
star	BG	411	667	2.9K	66.3M	537	846	7.03K	80M	630	970	5.53K	96.3M	731	1.16K	6.15K	104M
	SF	360	568	1.49K	35.4M	414	620	1.75K	25.1M	516	841	2.14K	32M	599	928	2.39K	36.2M
	WLE	1.05K	2.02K	13.7K	3.44B	1.29K	2.6K	17K	3.35B	1.45K	3.06K	18.8K	4.46B	1.68K	3.4K	21.4K	5.14B
	FS	1.17K	OOM	19.6K	OOT	1.5K	OOM	23.7K	OOT	1.82K	OOM	27.1K	OOT	2.06K	OOM	31.3K	OOT
cycle	BG	80	81	742	77.6M	142	174	1.44K	123M	281	495	2.61K	130M	225	798	2.34K	132M
	SF	197	319	605	73.9M	118	176	298	21.4M	102	144	424	105M	172	329	653	86.9M
	WLE	249	450	2.42K	3.06B	371	703	3.97K	4.4B	601	1K	5.1K	2.75B	467	840	4.44K	2.42B
	FS	660	OOM	6.04K	OOT	148	OOM	1.2K	OOT	346	OOM	3.62K	OOT	533	OOM	5.17K	OOT



(a) Varying graph density ($n = 10^6$) (b) Varying number of labels on CT



(c) Varying number of labels on FS (d) Varying number of labels on SPL

Figure 5: Impact of graph density and number of labels.

in Table 11. As we can observe, DLCR is still nearly one order of magnitude faster than P2H+, DLCR-BST is still around 2x faster than P2H+, and DLCR is nearly five order of magnitude faster than ARRIVAL on the real workloads. This is consistent with our experimental findings in Table 10 and demonstrates the effectiveness of our index scheme to handle real workloads.

Exp 6: Impact of graph density and number of labels . Next, we examine the impact of graph density and number of labels in graphs to query efficiency and update efficiency.

To test the impact of graph density, we adopt the Preferential Attachment (PA) model to generate graphs following [21]. We fix the number of nodes to be 1 million and the number of labels to be 8, and then vary the average degree of the graph to be {16, 32, 64, 128, 256}. As shown in Figure 5(a), with the increasing of density, both our update algorithms are taking a significantly reduced update cost. To explain, with denser graphs, the connectivity is getting better. Thus, the insertion or deletion of an edge will cause less effect to the overall reachability information. This further results in a reduced size of affected nodes in our update algorithms, providing improved update efficiency. In addition, the query efficiency gets a similar trend. To explain, with improved density, the index entries tend to help prune more paths, resulting in a reduced number of index entries, thus saving the query time.

We further test the impact of the number of labels in the graph. For datasets with a small number of labels, We vary the number of synthetic labels from 2 to 8 and only report the result on CT and FS datasets for the interest of space. As shown in Figures 5(b)-(c), the update cost of insertion and deletion on both CT and FS datasets are increasing. To explain, the smaller the number of labels is, the better the connectivity of the node pairs is (via fewer labels). Thus, similar to the graph density case, the insertion or deletion of an edge will cause less effect to the overall reachability information, contributing to an improved update efficiency. The query performance shows a similar trend. Again, with a small number of labels, the graph gains better connectivity and thus should improve the query performance as we observed in the experiment on the impact of graph density.

For datasets with a large number of labels (e.g., SPL), we vary the number of labels from 3000 to 7000, and examine the impact to our update algorithms. As we can observe, the performance of insertion, deletion, and query all do not change much when we increase the number of labels from 3000 to 7000. This phenomena is related to our index mechanism for large labeled graphs, where we map most labels to virtual labels. For example, we keep the top 4 most frequent labels to build the primary index and then we map the remaining labels to 4 virtual labels and build the secondary index. For instance, if we have 4004 labels. Then, we keep the top 4 most frequent labels as it is. For the remaining 4000 labels, we may randomly choose a set L_1 of 1000 labels and then map them to a new label l_1. In other words, for any edge whose label belong to L_1, we reset their label to l_1. By this strategy, we create a labeled graph with only 8 different labels: 4 primary labels and 4 virtual labels.

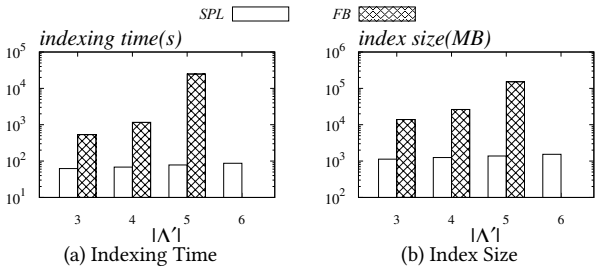
Note that the number of edges falling into each virtual label is roughly the same. Thus, the number of labels used for index construction (frequent labels and virtual labels) is the same for all settings. That is why the update cost is roughly the same when we vary the number of labels from 3000 to 7000. With a similar index, the query performance thus does not change much as well.

Exp 7: When incremental update is more efficient than reconstruction. Next, we conducted a new set of experiment to report the point when handling update incrementally is not efficient any more, in which case the update cost is the same as the cost of a re-construction of the index. In the Table 12, we report the ratio of the number m' of edges handled in the update over the number m of edges in the original graph when the incremental update incurs the same cost as an index re-construction.

As shown in Table 12, we can see that our update algorithms are very efficient and effective in most scenarios. For example, for

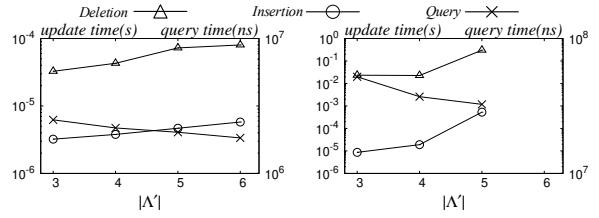
Table 12: Ratio of number m' of updated edges over m: When incremental update is the same as an index reconstruction.

Dataset	Deletion	Batch Deletion	Insertion	Batch Insertion
RB	11.3%	100%	61.2%	79.2%
AG	10.7%	26.4%	51.3%	53.1%
AX	0.0126%	0.163%	0.143%	1.89%
EP	21.4%	34.9%	39.5%	40.0%
SH	3.1%	14.8%	100%	100%
ND	1.49%	4.11%	6.75%	12.3%
BG	1.24%	30.5%	44.1%	45.3%
CT	0.0196%	1.58%	0.257%	2.93%
SF	3.74%	9.31%	48.2%	51.9%
WS	0.629%	1.72%	3.58%	9.17%
WG	0.215%	1.27%	2.2%	7.34%
YT	2.74%	1.28%	28.6%	32.5%
ZS	0.0152%	0.7%	0.656%	3.45%
SP	11.6%	20.4%	34.9%	35.1%
WL	18.6%	27.3%	46.3%	46.5%
DBP	0.615%	0.995%	3.75%	3.78%
WLE	18.9%	24.4%	47.5%	47.6%
T3W	1.93%	1.97%	35%	36.6%
TM	5.22%	9.56%	73.6%	74.5%
FS	0.88%	1.95%	72.5%	72.7%
SPL	1.47%	26.4%	59.2%	60.4%
FB	0.329%	1.16%	10.1%	11.0%



(a) Indexing Time (b) Index Size

Figure 6: Impact of $|\Lambda'|$ to indexing cost.



(a) SPL (b) FB

Figure 7: Impact of $|\Lambda'|$ to query and update.

batch insertion, at least half of the datasets can support around 10% of the edges for batch insertion, which is still more efficient than a re-construction. For batch deletion, 17 out of 22 of the datasets can support around 10% of the edges for batch deletion that is still more efficient than a re-construction. This demonstrates the high efficiency of our proposed update scheme.

Exp 8: Impact of $|\Lambda'|$. Recall that for large label graphs, we maintain a set Λ' of frequent labels and a same number virtual labels to generate the index. Next, we examine the impact of $|\Lambda'|$.

In particular, we conduct a set of experiments to examine the query performance, indexing cost, and update performance on the two graphs with large labels (SPL and FB). The results are shown in Figures 6-7. Notice that when $|\Lambda'| = 6$, the indexing time on FB dataset exceeds 24 hours and thus is omitted. As we can observe, the indexing cost, both the indexing time and index size, grows significantly when $|\Lambda'|$ increases from 3 to 6. In terms of update cost, both insertion and deletion algorithms incur a higher running cost when $|\Lambda'|$ increases. This is consistent to our observations in Figure 5, where a smaller number of labels tends to provide better connectivity, thus improving the update efficiency. Different from Figure 7, the query performance of DLCR actually improves when it grows from 3 to 6. To explain, with a larger $|\Lambda'|$, it can help prune more unreachable queries, thus improving the query efficiency. Our experimental results show that when we set $|\Lambda'| = 4$, it gains a good balance among query performance, indexing cost, and update performance. Thus, we set $|\Lambda'| = 4$ as the default setting.

Exp 9: Mixing workload of query and updates. Next, we examine the performance of DLCR under a stream of mixing workload. In particular, we vary the query/update (Q/U in Table 13) ratio from 0.05 to 0.8, and examine the performance of all competitors. Notice that since P2H+ is a static method and cannot handle updates in such a streaming manner, and thus is not included.

When the query/update ratio is 0.05, most of the operations are updates. In this case, ARRIVAL does not incur any additional burden to do the index updates while our DLCR needs to dynamically update the index. Even in such update-frequent scenarios, we can observe that our DLCR achieves the best performance on 13 out of 22 datasets. For the remaining 9 datasets (RB, AX, SH, ND, CT, WS, WG, ZS, and FB) that ARRIVAL is faster, if we check the accuracy of queries as shown in Table 7 in the revised manuscript, we can find that accuracy are very low for true queries on most datasets (RB 58.2%, AX 0.8%, ND 10.4%, CT 0.9%, WS 3.4%, WG 2%, ZS 1%, FB 84.5%). When the query/update ratio increases, our DLCR gains further better efficiency and dominates ARRIVAL on more datasets (18 out of 22 when query/update ratio is 50%). When the query/update ratio is 50%, our DLCR is still up to three orders of magnitude faster than ARRIVAL.

6 RELATED WORK

Label Constrained Reachability (LCR) Queries. The first work on LCR queries is proposed by [16]. 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 [21]. Compared with the landmark index-based algorithm[25], 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. Fan et al. present a theoretical study on what are doable and undoable for incremental graph algorithms [11]. [20] 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 batch. [10, 13, 22, 23]

Table 13: Average operation time of stream workloads in nanoseconds. ("OOT" indicates that this method is out of time during initializing, "OOM" indicates that this method is out of memory during indexing.)

Dataset	Q/U=0.05			Q/U=0.2			Q/U=0.35			Q/U=0.5			Q/U=0.65			Q/U=0.8		
	DLCR	DLCR-BST	ARRIVAL	DLCR	DLCR-BST	ARRIVAL	DLCR	DLCR-BST	ARRIVAL	DLCR	DLCR-BST	ARRIVAL	DLCR	DLCR-BST	ARRIVAL	DLCR	DLCR-BST	ARRIVAL
RB	11.1K	8.36K	**3.49K**	**6.48K**	7.41K	12.2K	**3.2K**	3.79K	13.3K	**2.57K**	2.98K	19.3K	**3.06K**	3.86K	27.2K	**2.22K**	2.73K	31K
AG	**5.42K**	5.94K	60.3K	**2.54K**	2.78K	205K	**1.61K**	1.77K	379K	**1.21K**	1.31K	581K	**619**	687	702K	**1.22K**	1.32K	830K
AX	422M	1.33B	**36K**	211M	680M	**151K**	263M	843M	**242K**	206M	744M	**385K**	154M	486M	**486K**	60M	214M	**560K**
EP	**34K**	44.4K	272K	**13.2K**	20.9K	1.09M	**18.1K**	27.8K	1.93M	**5.44K**	8.38K	2.82M	**5.31K**	8.37K	3.51M	**2.44K**	3.76K	4.24M
SH	27.9K	30.4K	**16.6K**	**2.42K**	3.45K	39.6K	**2.56K**	4K	68.2K	**1.32K**	1.82K	91.3K	**1.04K**	1.57K	136K	**540**	806	151K
ND	544K	1.03M	**296K**	**103K**	211K	845K	**100K**	217K	1.35M	**55.3K**	114K	2.34M	**65.3K**	136K	3.3M	**24.6K**	47.3K	4.14M
BG	**39K**	43.3K	540K	**2.97K**	4.25K	2.54M	**2.69K**	3.94K	3.94M	**7.25K**	11.1K	5.56M	**1.41K**	2.04K	7.26M	**1.08K**	1.54K	9.64M
CT	35.7M	99.8M	**65.7K**	38.8M	105M	**209K**	165M	583M	**378K**	47.2M	117M	**492K**	71.3M	172M	**633K**	10.4M	28.5M	**830K**
SF	**47.2K**	51.5K	383K	**4.89K**	7.49K	1.4M	**2.44K**	3.26K	2.42M	**1.84K**	2.54K	3.58M	**1.23K**	1.64K	4.02M	**504**	642	5.22M
WS	3.67M	8.57M	**786K**	**913K**	2.06M	2.99M	**1.01M**	2.46M	4.83M	**337K**	655K	7.12M	**64.9K**	124K	9.22M	**773K**	1.57M	11.3M
WG	4.04M	8M	**239K**	18.7M	40.3M	**1M**	**547K**	1.06M	1.74M	**159K**	328K	2.27M	**643K**	1.3M	2.91M	**105K**	200K	3.78M
YT	**1.09M**	2.39M	2.68M	**1.11M**	2.7M	11.4M	**947K**	2.17M	18.6M	**1.22M**	3.2M	26.3M	**92.9K**	242K	32.2M	**83.8K**	260K	46.3M
ZS	7.89M	19.5M	**159K**	363M	994M	**624K**	**360K**	848K	1.14M	14.6M	43.8M	**1.52M**	**432K**	1.04M	1.97M	32.6M	89.5M	**2.41M**
SP	**1.02M**	1.06M	1.78M	**29.3K**	49.4K	11.1M	**22.6K**	39K	22.6M	**80.4K**	138K	32.6M	**23.2K**	39.8K	39.8M	**7.97K**	13.8K	50.6M
WL	**209K**	304K	4.63M	**708K**	824K	24M	**199K**	223K	40.1M	**135K**	152K	56.3M	**232K**	242K	65M	**101K**	**101K**	86.3M
DBP	**430K**	673K	4.24M	**507K**	822K	12M	**1.2M**	1.59M	26.3M	**472K**	625K	32.7M	**181K**	308K	54.9M	**175K**	254K	64.5M
WLE	**283K**	383K	6.83M	**257K**	373K	39.7M	**513K**	559K	72.3M	**76.2K**	107K	109M	**83K**	105K	123M	**35.1K**	42.5K	151M
T3W	**1.08M**	1.11M	OOT	**672K**	996K	OOT	**246K**	418K	OOT	**352K**	526K	OOT	**122K**	192K	OOT	**51.4K**	99.2K	OOT
TM	**1.44M**	1.65M	OOT	**1.1M**	1.5M	OOT	**266K**	488K	OOT	**183K**	313K	OOT	**346K**	528K	OOT	**89.5K**	129K	OOT
FS	**14.6M**	OOM	OOT	**3.74M**	OOM	OOT	**2.15M**	OOM	OOT	**1.51M**	OOM	OOT	**1.16M**	OOM	OOT	**940K**	OOM	OOT
SPL	**44K**	68.3K	1.01M	**25.9K**	27.3K	3.05M	**45.5K**	46.7K	5.18M	**64.2K**	65.4K	14.7M	**82.7K**	83.5K	15.6M	**102K**	103K	14.9M
FB	475K	577K	**72.4K**	3.55M	3.51M	**153K**	5.86M	5.93M	**1.3M**	8.91M	8.75M	**1.05M**	12.5M	12.5M	**2.44M**	14.1M	13.8M	**4.99M**

propose algorithms for incrementally maintaining transitive closures on dynamic graphs. Nevertheless, these methods can not scale to billion-scale graphs which are shown by [17]. [6, 9, 24] propose methods to update the 2-hop labeling index. [15, 29, 30] present algorithms for performing updates on a reachability index. TOL [30] 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 AND FUTURE WORK

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. For future work, we plan to investigate devising efficient parallel algorithms for index construction and update 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] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 155–169.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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).

[30] 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.