Answering Billion-Scale Label-Constrained Reachability Queries within Microsecond

You Peng
The University of New South Wales
unswpy@gmail.com

Ying Zhang
The University of Technology Sydney
ying.zhang@uts.edu.au

Xuemin Lin
The University of New South Wales
lxue@cse.unsw.edu.au

Lu Qin
The University of Technology Sydney
lu.qin@uts.edu.au

Wenjie Zhang
The University of New South Wales
zhangw@cse.unsw.edu.au

ABSTRACT

In this paper, we study the problem of label-constrained reachability (LCR) query which is fundamental in many applications with directed edge-label graphs. Although the classical reachability query (i.e., reachability query without label constraint) has been extensively studied, LCR query is much more challenging because the number of possible label constraint set is exponential to the size of the labels. We observe that the existing techniques for LCR queries only construct partial index for better scalability, and their worst query time is not guaranteed and could be the same as an online breadth-first search (BFS).

In this paper, we propose novel label-constrained 2-hop indexing techniques with novel pruning rules and order strategies. It is shown that our worst query time could be bounded by the in-out index entry size. With all these techniques, comprehensive experiments show that our proposed methods significantly outperform the state-of-the-art technique in terms of query response time (up to 5 orders of magnitude speedup), index size and index construction time. In particular, our proposed method can answer LCR queries within microsecond over billion-scale graphs in a single machine.

PVLDB Reference Format:
You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Answering Billion-Scale Label-Constrained Reachability Queries within Microsecond. *PVLDB*, 13(6): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/3380750.3380753

1. INTRODUCTION

Graph is a ubiquitous structure representing entities and their relationships applied in many areas such as social networks, web graphs, and biological networks [28, 27, 11, 22, 17, 21, 23, 24, 25]. One of the most fundamental research problems on graphs is the reachability query, which checks if one vertex can reach another vertex or not in a graph. This problem seems simple and fundamental but very challenging due to the increasing size of graph data. In recent years, many research efforts have been devoted to efficiently support reachability queries on large graphs (e.g., [12, 14, 16, 31, 40, 52, 41, 30, 50, 53, 59]). Most of these techniques only consider the structure information of the graph and all edges are involved in the reachability queries. However, in many real-life applications, the graphs are edge-labeled, in which labels represent relationships between vertices. For instance, in the social network, we may have different relationships between two users such as "friendOf", "colleagueOf" and "relativeOf". Therefore, it is rather natural to limit the edge types (i.e., labels) in the reachability queries. In this paper, we study the problem of *Label-Constrained Reachability* (**LCR**) query. Given a source vertex s, a target vertex t and a subset L of the set of all edge labels ζ of the graph G, the LCR query will check if there is a path from s to t in G using only edges with labels in L.

Below are several motivating applications for the problem of the LCR query.

Social Networks. In a social network [20, 18, 19, 49, 47, 48, 35, 56, 39, 57, 38, 10], a vertex represents an entity (e.g., user, poster, organization) and there is an edge if two entities are related. Naturally, these relationships could be different types such as "friendOf", "supervisedBy", "parentOf", "Like", "Follow". Typical usage of an LCR query on a social network is to check if two persons are related by certain types of relationships. For instance, in the social network analysis (SNA) for counter-terrorism (e.g., [15]), a common practice is to check if two suspects are connected by some types of relationships (e.g., relative, friend, like).

Biological Network. As shown in [29], one of the most fundamental problems in system biology is to understand how metabolic chain reactions take place in cellular systems. A widely used network by biologists is the metabolic network, in which a vertex represents a compound, and an edge indicates one compound that could be transformed into another one through a certain chemical reaction. The label on an edge indicates the type of enzyme which controls the reaction (i.e., edge). By using LCR query, a metabolic network analyzer can quickly check if there is a pathway (i.e., chain of interactions) between two compounds through certain types of enzymes, as illustrated in Figure 1.

Knowledge Graphs. In Knowledge graphs, regular path queries are intensively studied (e.g., [4, 5, 6, 51]) and supported in practical graph query languages such as SPARQL 1.1, PGQL [44] and openCypher. LCR query is one of the

Proceedings of the VLDB Endowment, Vol. 13, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3380750.3380753

most important operators of the language of regular path queries. Particularly, an LCR query could be described by the regular expression $(l_1 \cup l_2 \cup ... \cup l_k)^*$, for constrained label set $L = \{l_1, .., l_k\}$, where $\cup$ is disjunction and $*$ is the Kleene star.

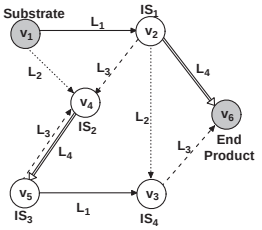


Figure 1: An example of a metabolic network where $|V| = 6$, $|E| = 9$, and $\zeta = \{L_1, L_2, L_3, L_4\}$. IS_i indicates the i_{th} intermediate substrate. L_i indicates the i_{th} enzyme.

In the above applications, efficient and scalable solutions for LCR queries on large scale graphs are critical for large graph analytics. Although a variety of efficient solutions have been proposed for the reachability query without the label constraint, these techniques cannot be trivially extended to efficiently support LCR queries because we need to build $2^{|\zeta|}$ indices for all possible label constraints.

Challenges. LCR is a well-studied problem and previous works face a big challenge when dealing with billion scale graphs. Although there are tons of papers studied 2-hop labeling and LCR queries, to the best of our knowledge, this is the first work to adopt this technique into LCR queries. When adopting the 2-hop labeling idea to the LCR query, the key challenges come from the huge number of the possible combinations of the labels. As a straightforward extension of 2-hop index to LCR query, we need to construct 2^l indices to consider every possible query label constraint where l is the number of labels. This is infeasible even for a very small l value due to the huge index size.

Novelty. To significantly reduce the index size while keeping the outstanding query efficiency of 2-hop index, we propose effective and efficient pruning rules and visiting order strategies to construct novel label-constrained (LC) 2-hop indexes, namely **P2H** and **P2H+**. In particular, we show that P2H+ has three nice properties: *soundness*, *correctness* and *minimal*, which make the query processing and index construction very efficient.

Our Contributions. Our principal contributions in this paper are summarized as follows:

- We propose three important properties for 2-hop indexing based techniques supporting the LCR queries. Based on three pruning rules and efficient construction algorithm, a label-constrained 2-hop index technique, namely **P2H**, is developed.
- We further enhance the performance by designing an advanced BFS search order and index entry construction order of the vertices. The resulting index is named **P2H+**, which satisfies all three good properties for label-constrained 2-hop index.
- Our comprehensive experiments demonstrate that our proposed approaches significantly outperforms the state-of-the-art technique LI+ in terms of query response time (up to 5 orders of magnitude speedup), index size and index construction time. To the best of our knowledge, this is the first work which can answer LCR queries over billion-scale graphs within a microsecond on a single machine.

Roadmap. The rest of the paper is organized as follows. Section 2 surveys important related work. Section 3 formally defines the problem, and describes the baseline solutions. Section 4 designs a pruned 2-hop based indexing. Two optimization techniques based on BFS search order and vertex order are proposed in Section 5, followed by the empirical study in Section 6. Section 7 concludes the paper.

2. RELATED WORK

In this section, we review closely related works.

2.1 Reachability Queries

There are some existing works on the problem of reachability queries(without label constraints). Xu and Cheng *et. al* give an excellent survey in [54]. An immediate online solution for reachability queries is BFS and [7] propose modern variants such as direction optimizing BFS(DBFS), which take $O(n + m)$ time, and have lower memory requirements than the full transitive closure(TC) of the graph, which requires $O(n^2)$ space. We use n and m to denote the number of vertices and edges of the graph, respectively. Also, many indexing methods have been proposed based on the idea of compressing the TC [42, 45], online search guided by precomputed indices [41, 30, 50, 53, 59], and labeling schemes [12, 14, 16, 31, 40, 52]. The 2-hop approach cannot be directly modified to LCR queries as the classical one does not consider labels. One straightforward modification of this approach is to build a separate graph G_{sl} for each possible set of labels sl. G_{sl} will contain only edges of G with labels in sl. However, this modified approach requires $O(2^{|\zeta|})$ space and time complexities as the number of different subsets of labels is $O(2^{|\zeta|})$, which is not scaled to large graphs. Besides the 2-hop labeling scheme, there are also a higher-compression indexing scheme namely, 3-hop proposed by Jin *et. al* in [32] .

2.2 Label Constrained Reachability and Path Queries

We then review the most closely related work on LCR and the current best-known method for LCR query evaluation.

Jin *et. al* [29]. This is the first work on LCR. In this work, two extremes for answering LCR queries are presented, namely, either BFS/DFS or building a full TC on the data graph. A tree-based index framework is presented in this work, which contains a spanning tree T and a partial transitive closure NT of the graph. Using T and NT, there is enough information to recover the full TC. Such approaches [33, 60], however, have failed to address the scalability problem, which is shown by follow-up work. Hence, we do not consider it further in our study.

Zou *et. al* [60] decomposes the input graph into strongly connected components (SCC's) $C_1, .., C_k$. The resulting index holds full reachability information. However, an obvious limiting factor is that it is not effective on graphs with a relatively large SCC. Hence, this method could not scale to large graphs and it is totally beaten by LI+ in [43]. Hence, we do not consider this method in our study.

Valstar *et. al* [43] proposes the state-of-the-art algorithm, namely LI+ for LCR queries. LI+ leverages landmark-based indexes for large graphs. To further speed up its practical performance, it also uses the non-landmarks index and pruning for accelerating false-queries. The main disadvantage of this method is that it is only a trade-off on landmarks. For

Table 1: The summary of notations

Notation	Definition
G, G^r	a given edge-labeled graph, its reverse graph
$e.src$, $e.dst$	source(target) vertex of an edge
$e.label$	edge's label
adj, adj^r	the adjacency list, the reverse adjacency list
$adj[l_i]$	adjacency neighbors with only label l_i
$adj[v][l_i]$	adjacency neighbors of v with only label l_i
$v \overset{\zeta}{\rightsquigarrow} u$, $v \overset{\zeta}{\not\rightsquigarrow} u$	v can(cannot) reach u with label set ζ
L_k	the constructed 2-hop index of k_{th} iteration.
$L_{in}[v]$, $L_{out}[v]$	the in(out) 2-hop index entries of v
$L_{in}[l]$, $L_{out}[l]$	the in(out) 2-hop index entries of for label l
$\zeta_0 \cup \zeta_1, \zeta_0 \setminus \zeta_1$	the union(difference) of two label sets

FULL-LI, the index cannot be built on most large graphs, while for LI+, the worst case will be even worse than DBFS for false queries due to the partial index. We aim to construct an index with full LCR information using less index size and construction time than LI+.

Edge-Disjoint Partitioning(EDP) [26] is recent progress on evaluating LCR queries where reasoning about path or distance is important [26, 8, 9, 58]. They mainly focus on the label-constrained shortest path(LCSP) problem. In [26], they partition graph by their labels and construct index in response to the queries received. We note that LCSP is more general than LCR, and constructing an index for such a problem is a more challenging and significantly different task. [55] proposed algorithms to solve the constrained shortest path queries in a time-dependent graph. In this paper, we do not consider these strategies further.

Recently Sarisht *et. al* [46] proposed a random-walk based sampling algorithm to answer regular simple queries on large labeled networks called ARRIVAL. Their algorithm could answer LCR queries. However, the main disadvantage of this method is that it only provides an approximate result. Also, Regular Label-Constrained Simple Path Queries are more general than LCR. Constructing such an index is more challenging and different. We do not consider these strategies in this paper.

3. PRELIMINARY

In this section, we first formally introduce the problem of LCR query following the definition in [43]. Then we introduce the-state-of-the-art indexing technique proposed in [43], namely *Landmark Indexing*(**LI+**). In Table 1, we summarize the important mathematical notations appeared throughout this paper.

3.1 Problem Definition

A graph $G = (V, E, \zeta, \lambda)$ is an edge-labeled graph in which V is a set of n vertices, ζ is a finite non-empty set of labels, and $E \subseteq V \times V \times \zeta$ is a set of directed labeled edges. For example, $e = \langle u, v, l\rangle \in E$ is an edge where u is the source vertex, v is the target vertex and l is the label of the edge. We denote λ as a mapping function: $E \to \zeta$ which maps edge to its corresponding labels, i.e., $\lambda(\langle u, v, l\rangle) = l$. When the context is clear, we use "graph" or "labeled graph" to represent the "edge-labeled graph" G in this paper.

A path P in graph G is a sequence $\langle v_0, e_0, v_1, ..., v_{p-1}, e_k, v_k\rangle$, for an integer $k > 0$, where $v_i \in V$ and $e_i \in E$ for every i. We denote the length of P as $|P|$. Furthermore, we say that P is an *L-path* or a label path if $\lambda(e_i) \in L$ for every $i \in [p]$.

In this paper, we say a vertex s can reach another vertex t through the label set L , denoted by $s \overset{L}{\rightsquigarrow} t$, if there is such an L-path connecting s to t, namely s-t L-path. Otherwise, we say s cannot reach t through the label set L, denoted by $s \overset{L}{\not\rightsquigarrow} t$. We say a label set $L \subseteq \zeta$ is a *minimal* label set connecting s to t if (i)$s \overset{L}{\rightsquigarrow} t$ and (ii)$s \overset{L'}{\not\rightsquigarrow} t$ for any label set $L' \subsetneq L$.

Based on this, we give the formal definition of label dominance as follows.

Definition 1. ***Label Dominance****: Given a source vertex s and a target vertex t, we say an s-t L_1-path with label set L_1 is dominated by another s-t L_2-path with label set L_2 if $L_2 \subsetneq L_1$.*

Then we have the definition of a minimal path as follows.

Definition 2. ***Minimal Path****: Given an s-t L-path P, we say P is a minimal path if there is no any s-t path P' where P is label dominated by P'; that is, we cannot find any s-t path P' with label set $L' \subsetneq L$.*

We give the formal definition of an LCR query in the following.

Definition 3. *An LCR query is a triple $(s, t, L) \in V \times V \times 2^\zeta$, where 2^ζ denotes the powerset of the label set ζ. If $s \overset{L}{\rightsquigarrow} t$, then this is a true-query. Otherwise, the query is said to be a false-query.*

Figure 1 is a running example of a metabolic network. The LCR query $(v_1, v_6, \{L_1, L_2, L_4\})$ is true while the LCR query $(v_1, v_6, \{L_1\})$ returns false. Furthermore, the query $(v_1, v_6, \{L_1, L_4\})$ is also true. Thus, the v_1-v_6 path with label set $\{L_1, L_2, L_4\}$ is not minimal due to the existence of v_1-v_6 path with label set $\{L_1, L_4\}$.

3.2 The State-of-the-Art

Now we introduce the state-of-the-art technique for LCR-query.

Landmark Index(LI^+) The state-of-the-art index-based algorithm is proposed in [43]. The key idea is to construct an index for partial landmarks by BFS from landmark one by one. When trying to insert a new index entry, it will compare to the existing index entries and only keep the minimal one. When dealing with an LCR-query, it answers the query with pruned BFS which uses the landmarks index. They also propose extended index techniques to accelerate the query e.g. index non-landmarks, pruning for accelerating false-queries. In this paper, we use their landmark-index with all techniques as a baseline.

3.3 2-Hop Cover Framework

The 2-hop cover technique has been extensively studied in the literature (e.g., [1, 13, 16, 3]). Our method follows the same framework. In particular, for each vertex v, we pre-compute a set of index entries denoted as $L(v)$ which includes $L_{in}(v)$ and $L_{out}(v)$, denoting *in-entries* and *out-entries* of the vertex v respectively. To answer an LCR-query from vertex s to vertex t with Label set L, we denote QUERY(s, t, L) as the answer, which is computed as follows:

$$QUERY(s,t,L) = \begin{cases} true & \exists u \in L_{out}(s) \text{ and } u \in L_{in}(t) \\ & \text{s.t. } s \overset{L}{\rightsquigarrow} u \overset{L}{\rightsquigarrow} t \\ false & Otherwise \end{cases}$$

4. LABEL-CONSTRAINED 2-HOP INDEXING

In this section, we first introduce two naive label-constrained (LC) 2-hop index methods in Section 4.1 and one of them will be used as a baseline in the performance evaluation. Then we highlight several desirable properties for the LC 2-hop index for LCR queries in Section 4.2. In Section 4.3, the query algorithm is proposed based on LC 2-hop index with soundness and completeness properties. In Section 4.4, we introduce our index construction algorithm for **P2H** index. In Section 4.5.1, we show the correctness for our proposed algorithms as well as complexity analysis.

4.1 Baseline Solutions

The 2-hop index technique[1] has been widely used in the reachability queries without label constraint. In this subsection, we introduce two naive solutions for LCR queries based on simple extensions of the existing 2-hop index techniques.

Given $|\zeta|$ labels, a straightforward label-constrained (LC) 2-hop index is to construct 2-hop index for every possible label set. Hence, we need to construct $2^{|\zeta|}$ indexes in total, which is infeasible in practice even for a small $|\zeta|$ and we do not consider this solution in our performance evaluation.

To avoid blowing up the index size, one possible way is to build the 2-hop index for each individual label, namely *one-label index*. As shown in Figure 1, graph G has labels L_1, L_2, L_3, L_4. Instead of constructing every possible label subsets, which is $2^4 = 16$ in total, we only construct four one-label index $\{(L_1), (L_2), (L_3), (L_4)\}$. In particular, for a label L_i, we consider all vertices and the edges with label L_i, which is named *one-label partition* of the graph. Then the classical 2-hop index can be constructed for each one-label partition of the graph.

Given vertices s and t with label set ζ_0, we can immediately ensure that $s \overset{\zeta_0}{\rightsquigarrow} t$ if there exists a label $L_i \in \zeta_0$ such that $s \overset{L_i}{\rightsquigarrow} t$ based on the one-label index of L_i. However, we *cannot* claim $s \overset{\zeta_0}{\not\rightsquigarrow} t$ even if we have $s \overset{L_i}{\not\rightsquigarrow} t$ for every $L_i \in \zeta_0$ because s may reach t through multiple labels. Similar to $LI+$, we have to keep on exploring the neighborhood of s due to the existence of "false negatives". In this paper, we call these methods *BFS-oriented* approaches since they need to continue to explore neighborhoods if there is no positive answer. In the *BFS-oriented* algorithm for LCR query, we can simply explore out-going edges like BFS when necessary, and test if each visited vertex can reach t with label constraint. Nevertheless, to speed up the computation, we propose a more efficient query processing method as described in Algorithm 1, which can explore the one-label index in BFS traverse. In particular, the while loop from Line 4 to 12 conducts a BFS traversal through every possible label in ζ_0. By $L_{out}[l]$, we denote the vertices in the out-edge index L_{out} in one-label index with label l. Note that, to accommodate the above query algorithm, we need to keep a reachability entry (u, v) of label L_i in both u and v.

REMARK 1. *The main disadvantage of the one-label index is that we have to resort to a BFS traverse in Algorithm 1 due to the existence of false negatives, which may incur expensive computing cost as demonstrated in our empirical study. This motivates us to design a new label-constrained 2-hop index without false negatives such that we can immediately confirm $s \overset{\zeta_0}{\not\rightsquigarrow} t$ based on the index entries on s and t alone.*

[1]To avoid possible ambiguity of "labeling index" and "label-constrained", we use "2-hop index technique" to present the "2-hop labeling index technique" used in the literature.

Algorithm 1 QueryNaive

```
1: procedure QueryNaive(L_k, src, dst, ζ_0)
2:     Stack S = {src}
3:     Set visited = {src}
4:     while S ≠ ∅ do
5:         v = S.pop()
6:         for label l ∈ ζ_0 do
7:             for u ∈ L_out[l] do
8:                 if u ∈ visited then
9:                     continue;
10:                visited.insert(u)
11:                S.push(u)
12:                if dst ∈ visited then return True
13:    return False
```

4.2 Properties of LC 2-Hop Index

We formally define the structure of LC 2-hop index, and then introduce several important properties that a good LC 2-hop index should have.

Label-Constrained (LC) 2-Hop Index ($\mathcal{L}$). An LC 2-hop index on a labeled graph $G = (V, E, \zeta)$ is a set of index entries $\{(u, \zeta_0)\}$ where u denotes a vertex and ζ_0 denotes a label set. For each vertex v, we use $L_{in}[v]$ and $L_{out}[v]$ to denote the in-going index entries (*in-entries* for short) and out-going index entries (*out-entries* for short). An index entry (u, ζ_0) in $L_{in}[v]$ implies that $u \overset{\zeta_0}{\rightsquigarrow} v$. Similarly, the index entry (u, ζ_0) in $L_{out}[v]$ implies that $v \overset{\zeta_0}{\rightsquigarrow} u$.

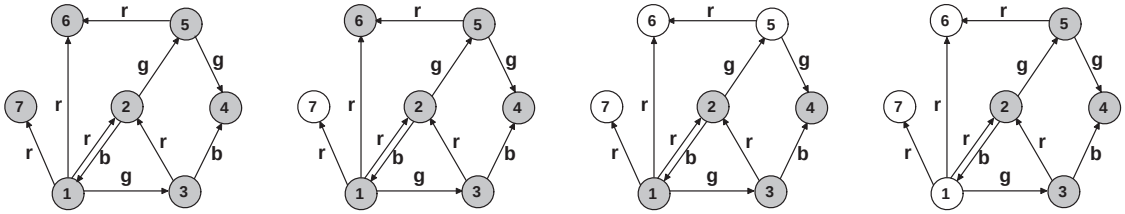
Below we show three properties to guide the construction of a good LC 2-hop index $\mathcal{L}$.

- **Soundness.** If there are two index entries $(v, \zeta_1) \in L_{out}[s]$ and $(v, \zeta_2) \in L_{in}[t]$ with $\zeta_1 \subseteq \zeta_0$ and $\zeta_2 \subseteq \zeta_0$, we have $s \overset{\zeta_0}{\rightsquigarrow} t$.
- **Completeness.** If $s \overset{\zeta_0}{\rightsquigarrow} t$, there must exist a vertex v such that $(v, \zeta_1) \in L_{out}[s]$ and $(v, \zeta_2) \in L_{in}[t]$, with $\zeta_1 \subseteq \zeta_0$ and $\zeta_2 \subseteq \zeta_0$.
- **Minimal.** For a vertex v, we say an entry $I = (u, \zeta_0) \in L_{out}[v]$ $(L_{in}[v])$ is *minimal* if I is not dominated by any other entries in $L_{out}[v]$ $(L_{in}[v])$; that is, there is no entry $I' = (u, \zeta_1) \in L_{out}[v]$ $(L_{in}[v])$ with $\zeta_1 \subsetneq \zeta_0$. We say an index entry $I = (u, \zeta_0) \in L_{out}[v]$ $(L_{in}[v])$ is **necessary** if there does not exist a vertex w s.t. $(w, \zeta_1) \in L_{out}[v](L_{in}[v])$ and $(w, \zeta_2) \in L_{in}[u](L_{out}[u])$ where $\zeta_1 \subseteq \zeta_0$ and $\zeta_2 \subseteq \zeta_0$. Finally, we say an LC 2-hop index $\mathcal{L}$ is **minimal** if every index entry is both minimal and necessary.

According to the definition, if an LC 2-hop index $\mathcal{L}$ is not *sound*, it may lead to *false positives*, i.e., LCR query (s, t, ζ_0) is false while the 2-hop index claims there is a s-t ζ_0-path. If $\mathcal{L}$ is *not complete*, there may exist *false negatives*; that is, we cannot safely claim $s \overset{\zeta_0}{\not\rightsquigarrow} t$ even if we cannot find any s-t ζ_0-path based on existing index entries in $L_{out}[s]$ and $L_{in}[t]$.

It is easy to say that the one-label 2-hop index proposed in Section 4.1 is *sound*. Therefore, we can immediately conclude that $s \overset{\zeta_0}{\rightsquigarrow} t$ if the index claims so. However, the index does not have the *completeness* property, and hence we have to explore the neighborhood of s if it cannot confirm $s \overset{\zeta_0}{\rightsquigarrow} t$. Suppose an LC 2-hop index satisfies both *soundness* and *completeness* properties, we can conclude $s \overset{\zeta_0}{\rightsquigarrow} t$ or $s \overset{\zeta_0}{\not\rightsquigarrow} t$ based on the index entries in $L_{out}[s]$ and $L_{in}[t]$ alone.

The *minimal* property implies that the index does not



(a) First BFS from vertex 1. We visited all the vertices. (forward and backward BFS).

(b) Second BFS from vertex 2. Vertex 7 is excluded.

(c) Third BFS from vertex 3. Only need to add 2-hop index entries to vertex 4.

(d) Fourth BFS from vertex 4. Only add 2-hop index entries to vertex 5.

Figure 2: Example of how pruning rule 1, 2, 3 works

include the redundant index entries where we say an entry is redundant if it can be derived by other index entries. The minimal property may lead to a small index size and hereby high query efficiency since the redundant entries are removed. In this paper, we aim to build an LC 2-hop index satisfying all these three properties.

4.3 Query Algorithm

Given a *sound* and *complete* 2-hop index, we denote $QUERY(s, t, \zeta_0)$ as the answer of s-t reachability query with label set ζ_0 and compute it as shown in Algorithm 2. In particular, we can claim $QUERY(s, t, \zeta_0) = true$ (i.e., $s \overset{\zeta_0}{\rightsquigarrow} t$) if there exist two entries $(u, \zeta_1) \in L_{out}[s]$ and $(u, \zeta_2) \in L_{in}[t]$ with $\zeta_1 \subseteq \zeta_0$ and $\zeta_2 \subseteq \zeta_0$. Otherwise, we have $QUERY(s, t, \zeta_0) = false$. Note that $L_{out}[s]$ ($L_{in}[t]$) consists a set of entries $\{(vertexID, label)\}$, and entries are ordered by vertex IDs. E.g., (1,r) (1,g), (2,r) are three entries in the $L_{in}[t]$. In this paper, we use $|L_{out}[s]|$ ($|L_{in}[t]|$) to denote the number of entries in the index.

Algorithm 2 Query Algorithm

```
1: procedure LCR Query(s,t,ζ0)
2:     for every index entry I_i in L_out[s] do
3:         if I_i.labels ⊆ ζ0 then
4:             for I_j in L_in[t] which has the same vertex as
   I_i.vertex do
5:                 if I_j.labels ⊆ ζ0 then
6:                     return True
7:     return False
```

4.4 Index Construction Algorithm

In this subsection, we introduce how to construct an LC 2-hop index, namely **Pruned 2-hop index** (**P2H** index for short). We conduct pruned BFS from $v_1, .., v_n$ from both forward and backward directions as shown in Algorithm 3 at Line 3 and 4, where L_k denote the $P2H$ index constructed after the k-th round. By L_n, we denote the $P2H$ index after processing all vertices (rounds).

Algorithm 4 illustrates how to conduct a pruned BFS starting from the vertex v_k. At Line 2, we use a set F to store current BFS frontiers. Line 4 to 14 is a while loop that continues the BFS search. The following three pruning rules are applied at Line 8, 10 and 13 within the while loop.

- **Rule 1**. We skip a vertex v if the index entries of v have been calculated (i.e., already been processed in Algorithm 3).
- **Rule 2**. We skip an index entry I if it can be derived from existing index entries; that is, the label-constrained reachability (LCR) information represented by the entry I is already available in the current index L_k. For instance, we do not need to keep an entry (u, ζ_0) for $L_{out}(v)$ if we have $LCR(v, u, \zeta_0) = true$ (i.e., Line 10 in Algorithm 4) based on the current P2H index L_k.
- **Rule 3**. As shown in Algorithm 5, we can remove the existing index entries which are dominated by the new index entry.

Figure 2 illustrates an example of our index construction algorithm. In this example, we consider the pruning rule 1, 2 and 3. Firstly, we explain the pruning rule 1. In the first BFS with starting vertex 1, all vertices are explored. In the second BFS, vertex 7 is skipped since vertex 1 will not be visited. Then for the third BFS, vertices $\{5, 6, 7\}$ are skipped. For the fourth BFS, vertices $\{1, 6, 7\}$ are skipped. Secondly, we explain the pruning rule 2. In the figure 2(a), we start with vertex 1. If 2 is explored, $(1, r,)$ will be inserted into in label of 2. Then $(1, rg)$ which is the index entry for $1 \to 3 \to 2$ will be pruned. At last, we introduce the pruning rule 3. Similar to pruning 2, if we explore $1 \to 3 \to 2$ first, and $(1, rg)$ will be inserted into in label of 2. Then we explore $1 \to 2$. In this case, the index entry $(1, rg)$ will be replaced by $(1, r)$ due to the dominance.

Table 2 shows in-entries and out-entries of the vertices in Figure 2 where all three pruning rules are applied. In the first forward BFS from vertex 1, vertices $\{2, 3, 4, 5, 6, 7\}$ are explored. Hence $(1, r), (1, g)$, $\{(1, rg), (1, rb)\}$, $(1, rg)$, $(1, r)$, and $(1, r)$ are inserted into in-entries of vertices $2, 3, 4, 5, 6$ and 7 respectively. For the first backward BFS from vertex 1, vertices 2, 3 are explored and $(1, b)$, $(1, rb)$ are inserted into out-entries of vertices 2 and 3 respectively. A similar procedure will be done for the following vertices. Note that indexed vertices will not be accessed in the following procedure. For the pruned BFS procedure from vertex 5, only $(5, r)$ is inserted into out-entries of vertex 6 (i.e., $L_{out}[6]$). We do not insert it into vertex 4's out-entries. Observe that for each vertex u, we only need to add (v, ζ_v) when v is ranked lower than u. After constructing these index, given a query $LCR(7, 5, rg)$, that is whether the vertex 7 can reach 5 with label set $\{r, g\}$, we can find $(1, rg)$ in 5's in-entries and $(1, r)$ in 7's out-entries. Hence the answer is true for this query.

Table 2: Index entries fo Figure 2

ID	in-entries	out-entries
1	-	-
2	(1,r)	(1,b)
3	(1,g)	(1,rb),(2,r)
4	(1,rg),(1,rb),(2,g),(3,b)	-
5	(1,rg),(2,g)	(4,g)
6	(1,r),(2,rg),(5,r)	-
7	(1,r)	-

4.5 Analysis

Algorithm 3 Construct 2-hop index by Pruned BFS

```
1: procedure ConstructIndex(Graph G)
2:     for k = 1, 2, ...n do
3:         L̂_k = PrunedBFS(adj, v_k, L_{k-1})
4:         L_k = PrunedBFS(adj^r, v_k, L̂_k)
5:     return L_n
```

Algorithm 4 Pruned BFS from v_k to create L_k

```
1: procedure PrunedBFS(AdjacencyLIst Adj, vertex v_k, index L_{k-1})
2:     Set F = {(v_k, ∅)}          ▷ keeps track of the current frontier.
3:     L_k = L_{k-1}
4:     while F ≠ ∅ do
5:         v = F.pop()
6:         for every edge e in adj[v.node] do
7:             l_new = e.label + v.labels
8:             if e.dst in v_1, .., v_{k-1} then
9:                 Continue;                ▷ already indexed
10:            if QUERY(v_k, e.dst, l_new, L_k) = True then
11:                Continue;                ▷ already indexed
12:            else
13:                TryToInsert(L_k, v_k, e.dst, l_new); ▷ Swap v_k and e.dst when conduct reverse BFS
14:                F.insert({e.dst, e.labels + v.labels})
15:    return L_k
```

In this subsection, we analyze the correctness and complexity of **P2H**.

4.5.1 Proof of Correctness

In the following, we show that **P2H** index constructed in Algorithm 3 is correct; that is, $QUERY(s, t, \zeta_0)$ is correct for any source vertex s, target vertex t, and label set $\zeta_0 \subseteq \zeta$.

In this paper, we say an s-t L-path P is *stored* in L_k if its corresponding LCR information $LCR(s, t, L) = true$ can be derived from the index L_k.

Theorem 1. *For any $0 \leq k \leq n$ and for any pair of vertices s and t with label constraint set ζ, every L-path containing vertex v_i $(i \leq k)$ is stored in L_k.*

Proof. We prove the theorem by mathematical induction on k. When $k = 0$, it is obvious that the theorem is true. Now we assume it holds for $0, 1, .., k-1$,and then prove it also holds for k.

We prove this by contradiction. We assume there exists an L-path which contains v_j and not stored in L_k with $j \leq k$. Firstly, if $j < k$, then it must be in $L_j \subset L_k$ which contradicts the previous assumption. Hence, $j = k$.

Since this L-path is not in L_{k-1}, for any vertex v_i in this L-path, we have $k \leq i$. Regarding our three pruning rules in Algorithm 4, this L-path will be explored. Since it is not stored in L_k, any vertex in this L-path is not chosen in the previous stage (since for any vertex v_i, we have $k \leq i$) and explored in k_{th} BFS, it will be inserted into L_k. This contradicts the assumption. □

Algorithm 5 TryToInsertIndex

```
1: procedure TryToInsert(L_k, src, dst, ζ_0)
2:     if {src, dst, ζ_0} is a subset of some index entry l_i {src, dst, ζ_i} ∈ L_k then
3:         Repalce l_i with {src, dst, ζ_0} return False
4:     else if {src, dst, ζ_0} is a superset of some index entry l_i {src, dst, ζ_i} ∈ L_k then return False
5:     else
6:         L_k.insert({src, dst, ζ_0})
7:     return True
```

As a corollary, our method is proved to be correct by instantiating the theorem with $k = n$.

Corollary 1. *Any L-path in Graph G=$\{V, E, \zeta\}$ will be stored in L_n.*

The *completeness* of L_n (i.e., P2H index) is immediately based on corollary 1. The proof of *soundness* is trivial since we insert every LCR information after confirmation. Therefore, the **P2H** index constructed is both *sound* and *complete*.

4.5.2 Space and Time Complexity

For each vertex, it will store at most $n - 1$ vertices and at most $2^{|\zeta|}$ different label sets for each vertex. Hence, the space complexity is $O(n^2 2^{|\zeta|})$.

For the query (s, t, ζ_0), it takes $O((|L_{in}[s]| + |L_{out}[t]|) \times |\zeta_0|)$ time where $|L_{in}[s]|$ $(|L_{out}[t]|)$ denotes the number of (vertex, label) entries in label of s (out label of t) and $|\zeta_0|$ is the size of the query label constraint. Particularly, as entries are already sorted by vertex IDs, it takes at most $O(|L_{in}[s]| + |L_{out}[t]|)$ time to find entries $\{(v, \zeta_i)\}$ such that $v \in L_{in}[s]$ and $v \in L_{out}[t]$. Meanwhile, it takes at most $O(|\zeta_0|)$ time to check if $\zeta_i \subseteq \zeta_0$. Thus, the time complexity of Algorithm 2 is $O((|L_{in}[s]| + |L_{out}[t]|) \times |\zeta_0|)$.

As for the index construction process, it will take $O(m * n^2 * 2^{2*|\zeta|})$. For each pruned BFS, each edge will be touched at most $O(2^{|\zeta|})$ since the number of different paths with different labels that can reach an edge can be $2^{|\zeta|}$. Also, for each visited edge, the label checking cost could be $O(n * 2^{|\zeta|})$ and we need n iterations.

5. BFS SEARCH ORDER AND VERTEX ORDER STRATEGIES

5.1 Motivation

As shown in Section 4.5.1, the P2H index has both *soundness* and *completeness* properties, which enable us to quickly check the LCR queries based on the index entries from source and target vertices alone. Nevertheless, P2H index does not guarantee the *minimal* property. In this section, we further enhance the performance by developing novel Breadth-First Search order and vertex accessing order strategies. The new CL 2-hop index technique, namely P2H+, has all three desirable properties. With the minimal property, P2H+ technique may come up with a smaller index size and hereby faster query response time.

5.2 Breadth-First Search Order

In Figure 3, we show that the BFS search order in Algorithm 4 may affect the size of the resulting index. Suppose we start BFS from vertex 1 with current frontiers $\{2, 3, 4, 9\}$. If we first visit the edge (9,5,r), then the index entries resulted from following edges $\{(4, 5, g), (3, 5, b), (2, 5, w)\}$ will be pruned because of the index entry (1,r) in vertex 5's in-entries. However, if we follow another order, say (4,5,g), (3,5,b), (2,5,w) and (9,5,r). Both entries $(1, r)$, $(1, rw)$, and $(1, rb)$ will be kept in $L_{in}[5]$, which is not minimal according to the definition.

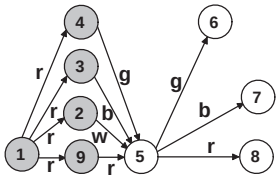


Figure 3: Motivation of BFS search order

Motivated by this example, we propose a novel BFS Search Order in Algorithm 4 to ensure that once an index entry is inserted, it will not be pruned by the following inserted index entries. Then, we will show that this advanced order strategy will ensure the minimal property of the resulting index.

When we conduct a pruned BFS from vertex v, we aim to only explore the minimal path and then the constructed index is minimal since any index entry could not be pruned by any other index entries including the existing ones and the ones inserted later. In the following, we present an advanced Pruned BFS to get an LC 2-hop index with the *minimal* property. We define a new array $adj[v][\zeta_v]$ to denote the out-neighbor of vertex v with edge label ζ_v. Algorithm 6 is similar to Algorithm 3. The difference is that in Algorithm 6, one iteration in a BFS is divided into two parts. One is at Line 5 which only uses current labels in BFS. As shown in the example in Figure 3, if we start from vertex 1 and current frontiers are $\{(4,r),(3,r),(2,r),(9,r)\}$, then for $(9,r)$, we will only use BFS with edges in label r in Line 5. If $\zeta = \{r,g,b\}$, then in Line 6 will touch edges with label $\zeta \setminus r = \{g,b\}$. That means in this example, Algorithms 7 will only explore labels which are the same as the current entry e.g. r since (9, r) is the current entry. Algorithms 8 will explore edges with labels $\zeta \setminus r = \{g,b\}$. In a word, the main difference is that Algorithm 7 only uses current labels at Algorithm 7 Line 5 while Algorithm 8 will include the remaining available labels at Algorithm 8 Line 4.

Algorithm 6 Pruned BFS from v_k to create L_k

```
1: procedure PrunedBFS(AdjacencyLIst Adj, vertex v_k, index L_{k-1})
2:     Set F = {(v_k, ∅)} ▷ keeps track of current frontier.
3:     L_k = L_{k-1}
4:     while F ≠ ∅ do
5:         F,L_k = BFSWithCurLabels(adj, F, L_k)
6:         F,L_k = BFSPlusOneLabel(adj, F, L_k) ▷ Similar for reverse BFS.
7:     return L_k
```

5.3 Minimal Property

In this subsection, we show that the new index constructed by the advanced BFS Search Order has the *minimal* property. Before prove the minimal property, we introduce a lemma.

Lemma 1. *For two label paths P_1 and P_2 with source, target vertex and label set $\{s_1,t_1,\zeta_1\}$ and $\{s_2,t_2,\zeta_2\}$ respectively, P_1 is dominated by P_2 if and only if (1) $s_1 = s_2, t_1 = t_2$; (2) $\zeta_2 \subsetneqq \zeta_1$; and (3) $|\zeta_2| < |\zeta_1|$.*

The proof is trivial. For condition 1, it ensures these two paths have the same source and target vertices. Condition 2 is the definition of dominance and condition 3 could be derived by condition 2. Condition 3 will be used in our new BFS Search Order.

Algorithm 7 BFSWithCurrentLabels

```
1: procedure BFSWithCurLabels(AdjacencyLIst Adj, Frontier F, index L_k)
2:     Set ResultF = F
3:     while F ≠ ∅ do
4:         v = F.pop()
5:         for every label l_0 in v.labels do
6:             for every edge u in adj[v.node][l_0] do
7:                 l_new = u.labels ∪ v.labels
8:                 QR = QUERY{v_k, u.dst, l_new, L_k}
9:                 if u.dst in v_0, .., v_{k-1} or QR then
10:                    continue;                ▷ already indexed
11:                else
12:                    L_k[u.dst]_in.insert(v_k, l_new)
13:                    F.insert({u.dst, l_new})
14:                    ResultF.insert({u.dst, l_new})
15:    return ResultF, L_k
```

Algorithm 8 BFSPlusOneLabel

```
1: procedure BFSPlusOneLabel(AdjacencyLIst Adj, Frontier F, index L_k)
2:     Set ResultF = ∅
3:     for Every item v in F do
4:         for every label l_0 in ζ \ v.labels do
5:             for every edge u in adj[v.node][l_0] do
6:                 l_new = u.labels ∪ v.labels
7:                 QR = QUERY{v_k, u.dst, l_new, L_k}
8:                 if u.dst in v_0, .., v_{k-1} or QR then
9:                     continue;                ▷ already indexed
10:                else
11:                    L_k[u.dst]_in.insert(v_k, l_new)
12:                    ResultF.insert({u.dst, l_new})
13:    return ResultF, L_k
```

Theorem 2. *For any index entry (v,u,ζ_0) in L_n, the L-path (v,u,ζ_0) is minimal.*

Proof. We first prove the correctness of this method. It is similar to Theorem 1. Then we prove the minimal property. According to Lemma 1 condition 1, we only need to discuss the process for some vertex v. For a newly inserted index entry with (s,t,ζ_i), it cannot be dominated by any existing index entry since our algorithm will check this domination. Hence, we only need to prove that existing index entry will not be dominated by newly inserted index entry and then we will get a minimal index. When a new index entry (s,l_i) or (t,l_i) is inserted into $L_{in}[t]$ or $L_{out}[s]$. According to the BFS search order, all the inserted index entries with the same starting vertex, we denote as (s,l) or (v,l) will meet the condition $|l| \leq |l_i|$. This means that inserted index entry will not be dominated by newly inserted index entry due to Lemma 1. □

Theorem 3. *For any vertex v, and for any index entry $(u,\zeta_0) \in L'n[v]$, there is an LCR information (s,t,ζ_1) such that, if we remove index entry (u,ζ_0) from $L'_n[v]$, we cannot answer the LCR query (s,t,ζ_1) correctly.*

Proof. Let $v_i \in V$ and $(v_j,\zeta_j) \in L'_n[v_i]$. This implies $j < i$ due to our pruning rule 1 in the index construction. We show that if we remove (v_j,ζ_j) from $L'_n[v_i]$ then we cannot answer the LCR query (v_i,v_j,ζ_j) correctly. Due to our index construction process, the result for LCR Query(v_i,v_j,ζ_j) should be *true*. Due to theorem 2, if we remove (v_j,ζ_j) from $L'_n[v_i]$, there will not exist direct LCR information, e.g.

(v_j, ζ_{j1}) such that $\zeta_{j1} \subsetneqq \zeta_j$ in $L'_n[v_i]$. There does not exist (v_i, ζ_i) in $L'_n[v_j]$ since $j < i$. Then the following claim could prove the minimal property of our index: For any $k \neq j$, either (i) $(v_k, \zeta_k) \notin L'_n[v_i]$ or $(v_k, \zeta_k) \notin L'_n[v_j]$ holds, or (ii) there does not exist $(v_k, \zeta_{k1}) \in L'_{out}[v_i](L'_{in}[v_i])$ or $(v_k, \zeta_{k2}) \in L'_{in}[v_j](L'_{out}[v_j])$ s.t. $\zeta_{k1} \subset \zeta_j$ and $\zeta_{k2} \subset \zeta_j$.

Suppose $k < j$ and we assume that (ii) does not hold. Then, (i) must hold otherwise the *j-th* BFS will prune vertex v_i and $(v_j, \zeta_j) \notin L'_n[v_i]$ which contradict the assumption that $(v_j, \zeta_j) \in L'_n[v_i]$. Then we suppose $k > j$ and assume that (ii) does not hold. Then v_k will be in the vertex set of the minimal L-path between v_i and v_j. That implies $(v_j, \zeta_j) \in L'_j[v_k]$, thus the *k-th* BFS prunes vertex v_j, leading to $(v_k, \zeta_k) \notin L'_n[v_j]$. □

Theorems 2 and 3 indicate that every index entry in P2H+ index constructed by Algorithm 6 is both necessary and minimal. Therefore, P2H+ index has the minimal property by utilizing the advanced BFS search strategy. It is immediate that P2H+ index also has the *soundness* and *completeness* property since P2H+ simply enforces the BFS visiting order compare to P2H.

5.4 Vertex Ordering Strategies

5.4.1 Motivation

In our index construction algorithm, we build up the index entries for vertices following a particular order. Although this order does not affect three properties of the resulting index, it is crucial for the final performance of the index as shown in the empirical study. Thus, in addition to the visiting order in each BFS search of Algorithm 4, we further consider the access order of the vertices in Algorithm 3, i.e., order of vertices for index entries construction. The key idea to prioritize the "central" vertices which could cover more LCR information at an earlier stage. Vertex order is a crucial problem and widely discussed in the previous 2-hop labeling index for shortest path distance in [3]. Nevertheless, 2-hop labeling for LCR problem is different from the traditional one. In this problem, we need to find "central" vertices that can prune LCR information as much as possible. Different from shortest path distance pruning, LCR information pruning needs to compare the dominance of the label set.

5.4.2 Vertex access order strategies

DEGREE: The higher degree a vertex has, the earlier we build index entries on it. The insight behind this strategy is that a high degree vertex could have a strong connection to many other vertices. Then it is more likely to prune more index entries. When dealing with edge-labeled graph, one straightforward modification is to give each label type a weight based on the percentage of this type of edges in all edges. In our initial experiments, these two strategies have very similar performance since vertex with a high degree seems more likely to have high weighted label types. Hence, in this paper, we only consider the classical degree order strategy; that is, build index entries for vertices following the descendant order of their degrees.

SIGNIFICANT PATH: It is well known that the computation of betweenness is very cost expensive. Inspired by the significant path based node order strategy in [37, 2], we proposed a similar node order strategy based on minimal label path. The philosophy of this proposed node order is that "*adaptively accumulate node importance information during pruned breadth first search*". We choose an important vertex as the start node v (e.g., the vertex with the highest degree) and build the minimal label path tree rooted at v. We use a *count* array to record the time a vertex being touched during the building process[2]. Note that this array will only be initialized at the first vertex, that is to say, this information will be accumulated in the following stage. A vertex v with high count$[v]$ is likely to have a high betweenness value. Due to the power of our proposed pruning rules, we only choose the first $1250 + \sqrt{n}$ vertices based on the count array. For i_{th} pick, we will choose one neighbor of $v_{i-1}((i-1)_{th}$ *pick*) which has the highest count value. Otherwise, we pick vertex following the degree order strategy.

5.5 Space and Time Complexity

The difference between **P2H** and **P2H+** is the vertex search order. Hence, the worst case for construction part is the same. For the preprocessing part, **P2H+** needs a new adjacent list which only takes $O(m)$ space and time complexity. Based on this analysis, the space and time complexity of **P2H+** index are the same as **P2H** index, which is shown in Section 4.5.2.

5.6 Dealing With Large Number of Labels

In some applications such as knowledge graph, the number of labels $|\zeta|$ might be large. We cannot directly apply our technique in these scenarios because, although we already significantly reduce the index size compared to existing exact the state-of-the-art solutions, our approach may still be sensitive to the number of labels due to the rapid growth of $2^{|\zeta|}$. In this subsection, we investigate how to extend our solution to handle graphs with a larger number of labels.

It has been widely observed that the frequency of the labels in real-life graphs usually follows the power-law distribution. This motivates us to consider the high frequent labels and low frequent labels in a different way, where two LCR indices are constructed as follows.

Primary Label Index We build LCR index on the w most frequent labels, denoted by ζ_f. If query label set ζ_0 is the subset of ζ_f, we can immediately answer the query based on primary label index since all relevant labels are explicitly or implicitly indexed by the primary label index. Given the power law distribution of the label frequency, the majority of the queries will fall in this category. In case the query label constraint ζ_0 is not a subset of ζ_f (i.e., the most frequent w labels), if Query$(s, t, \zeta_0 \cap \zeta_f)$= True, then the result is also True for Query(s, t, ζ_0).

Secondary Label Index We build LCR index on w labels: $w/2$ most frequent labels and $w/2$ virtual labels. Specifically, we remove the $w/2$ most frequent labels and evenly partition the remaining labels into $w/2$ groups according to their frequency, where each group is represented by a virual label, i.e., the labels with the same virtual label are regarded as the same in the secondary label index. When the query result is false on the secondary label index, we could return a false result immediately. Otherwise, we need to further explore the neighbors of s in a recursive way where both primary label index and secondary label index might be employed, if the current secondary label indices on s and t return true.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness and efficiency of proposed techniques on comprehensive experiments.

[2]Our index construction process for each vertex will return a minimal label path tree.

Table 3: Statistics of Datasets. K indicates 10^3. M indicates 10^6. B indicates 10^9.

Name	Dataset	$\|V\|$	$\|E\|$	$\|\zeta\|$	Synthetic Labels
RT	robots	1.4K	2.9K	4	
ADG	advogato	5.4K	51K	4	
AX	arXiv	34K	421K	8	√
EPIN	epinions	131K	840K	8	√
SHS	StringHS	16K	1.2M	7	
ND	NotreDame	325K	1.4M	8	√
BG	BioGrid	64K	1.5M	7	
CT	Citeseer	384K	1.7M	8	√
SFC	StringsFC	15K	2.0M	7	
WS	webStanford	281K	2.3M	8	√
WG	webGoogle	875K	5.1M	8	√
YT	Youtube	15K	10.7M	5	
ZH	zhishihudong	2.4M	18.8M	8	√
SP	socPokec	1.6M	30M	8	√
WL	wikiLinks	3.0M	102M	8	√
WPE	WikipediaEng	18M	172M	8	√
WLE	WikiLinksEng	12M	378M	8	√
T3W	Twitter(3W)	41M	1.5B	8	√
TM	Twitter(MPI)	52M	2.0B	8	√
FS	Friendster	68M	2.5B	8	√
SPL	socPokecL	1.6M	30M	7513	√
FB	Freebase	14.4M	106M	772	

6.1 Experimental Settings

Algorithms We compare proposed algorithms with baseline solutions.

- **LI+**.The state-of-the-art landmark index-based algorithm in [43] with all the optimization techniques. Following the same settings in [43], the number of landmarks k is set to be $1250 + \sqrt{n}$ and the budget b per non-landmark vertex is set to be 20, where n is the number of vertices in the input network.
- **NP2H**. One-label 2-hop index algorithm for LCR queries proposed in Section 4.1.
- **P2H**.Our proposed LC 2-hop index algorithm for LCR queries which is introduced in Section 4.4.
- **P2H+**.Our proposed LC 2-hop index algorithm with advanced BFS Search Order for LCR queries which is introduced in Section 5.2 [3]. For datasets whose label size is larger than 10, we will use methods in Section 5.6 to address it where $w = 12$.

Datasets In this part, the datasets settings is similar to [43] and we also add some large graphs. Table 3 shows the important statistics of real graphs used in the experiments. Most of these graphs are from either SNAP [36] or KONECT [34]. Some graphs have natural edge labels. For those graphs without edge labels, we use the method the same as synthetic data, while the number of labels $|\zeta|$ to 8 and the parameter α to 1.7.

Settings In experiments, all programs are implemented in standard c++11 and compiled with g++4.8.5. The code of LI+ algorithm is provided by authors in [43]. Kindly thanks for authors to provide their code and datasets. All experiments are performed on a machine with 20X Intel Xeon 2.3GHz and 768GB main memory running Linux(Red Hat Linux 7.3 64 bit).

QueryGeneration. For each dataset, we generate three query sets, using $|L|/4$, $|L|/2$ and $|L| - 2$ labels or (2, 4, 6) when $|L|$ is too large. Each query set consists 1,000 true-queries and 1,000 false-queries. We use the same generation strategy as [43] and thanks for the authors for providing code for that.

[3] **P2H+** uses degree order which is the same as LI+ for the sake of comparison fairness. The effect of node order will be compared in Table 8

Exp 1: performance on real graphs

Table 4: Indexing time(IT) in seconds, and index size(IS) in megabytes for real datasets. In this table, "-" indicates that the method timed out on this dataset and "K" indicates 10^3.

Name	LI+		P2H		P2H+		NP2H	
	IT	IS	IT	IS	IT	IS	IT	IS
RT	0.15	4	0.01	0.61	0.01	0.59	0.01	0.76
ADG	4	135	0.15	3	0.12	3	0.07	4.47
AX	95	551	350	445	303	404	6.0	106
EPIN	284	2.9K	27	77	9	68	3.98	76.4
SHS	18	556	5	38	4	37	1.2	40.8
ND	398	2.4K	35	263	24	236	8.5	233
BG	56	1.1K	6	48	4	43	1.7	55.9
CT	79	562	312	1.3K	171	1K	46.6	777
SFC	26	515	7	56	7	55	1.8	58.5
WS	1K	7.8K	67	413	50	361	11.3	320
WG	6.1K	34K	547	1.4K	378	1.2K	301	744
YT	4.7K	335	1K	370	1.2K	365	232	380
ZH	4.2K	9.4K	15K	15K	10K	14K	167	2.7K
SP	26K	155K	3.9K	2.3K	1.1K	1.6K	312	1.9K
WL	61K	216K	5.5K	4.5K	1.9K	3.8K	1K	4.3K
WPE[4]	88K	2.4K	18K	13K	5.5K	10K	2.2K	12K
WLE	-	-	-	-	13K	14K	4.3K	16K
T3W	-	-	-	-	63K	55K	28K	63K
TM	-	-	-	-	79K	71K	38K	71K
FS	-	-	-	-	125K	93K	58K	117K
SPL	-	-	-	-	6K	4k	-	-
FB	-	-	-	-	54K	94K	-	-

In our first experiment, we compare our proposed algorithms with NP2H and LI+. First of all, for the indexing time and index size, P2H and P2H+ beat LI+ in the most datasets except ZH, CT, AX(indexing time) and YT(index size). However, in these datasets, our query performance is about one or two orders faster than LI+. This shows our algorithms could achieve better scalability than LI+ when dealing with large graphs. The key reason is LI+ need to store all LCR information for all chosen landmarks while our algorithm could pruned most of them by existing index entries. Also, **P2H+** develop novel vertices orders and BFS search orders which could futher reduce the index size. When we compare **P2H** and **P2H+**, it is shown that the index time of **P2H+** is faster than **P2H** except for the dataset youtube. The index size of **P2H+** is always smaller than **P2H** due to the minimal property of **P2H+**. It is reported that when the input graph gets larger, **P2H+** shows that it is more scalable than **P2H** e.g. on the datasets socPokec and wikiLinks(fr), **P2H+** only needs one third index construction time of **P2H** and can save 45% and 20% index size respectively.

For query time performance, our algorithms could achieve about two(true-queries) or three(false-queries) orders of magnitudes than LI+ on average and up to five orders when answering false-queries. Compared **P2H** with **P2H+**, **P2H+** shows a similar performance with **P2H**. One interesting result is that **P2H+** shows better performance on false query set, since false query needs to traversal all out labels of source vertex and in labels of target vertex and **P2H+** has a minimal index. **P2H** beat **P2H+** in some cases since minimal index does no mean the optimal one. It is also shown in Table 5 that the larger the label size is, the better **P2H+** performances than **P2H**. As for NP2H,

[4] k=2 and b=0

Table 5: Speed-ups of **P2H** and **P2H+** over LI+ for each of the real datasets in true query sets. For LI+, the average query time is given in microseconds. In this table, "-" indicates that the method timed out on this dataset.

	$\lfloor\zeta/4\rfloor$ or 2				$\lfloor\zeta/2\rfloor$ or 4				$\lfloor\zeta-2\rfloor$ or 6			
Name	**P2H**	**P2H+**	LI+ (µs)	NP2H	**P2H**	**P2H+**	LI+ (µs)	NP2H	**P2H**	**P2H+**	LI+ (µs)	NP2H
RT	1.25	**1.43**	0.1	0.2µs	2.38	**10.17**	0.1	66µs	3.33	**10**	0.1	20µs
ADG	**6.79**	5.94	0.24	0.004µs	**10.75**	8.82	0.23	4.8µs	**8.98**	8.71	0.23	3.4µs
AX	**25.5**	20.6	6.4	12ms	25.4	**51.4**	4.1	19ms	46.6	**60.3**	2.05	16ms
EPIN	15.15	**17.24**	1.9	337µs	**20.01**	19.67	1.13	0.37µs	**33.88**	32.89	1.34	0.30µs
SHS	**73.31**	59.51	2.84	0.19µs	**268.15**	232	4.13	25.4µs	**58.96**	33.10	0.38	43.3µs
ND	193.53	**222.03**	25.9	26µs	131.52	**154.24**	8.39	17µs	27.46	**30.36**	3.45	1.9ms
BG	280.91	**404.38**	27.4	0.26µs	183.78	**369.34**	7.6	204ms	81.28	**290.14**	4.99	19.4ms
CT	16.9	**24.4**	4.4	3.4ms	16.8	**30**	4.2	15.3ms	20	**45.8**	3.6	23.7ms
SFC	**157.21**	98.70	4.4	0.14µs	**43.73**	35.38	0.78	129µs	**294.74**	199.66	4.95	265µs
WS	23.02	**32.34**	2.27	29µs	199.07	**311.48**	9.16	1.1ms	46.79	**55.16**	2.10	2.9ms
WG	**35.86**	31.14	7.57	0.6µs	35.11	**44.00**	9.59	24ms	30.52	**30.84**	6.42	44ms
YT	**100.37**	100.14	90.55	2.1µs	109.40	**121.74**	87.63	59µs	120.89	**132.96**	62.84	12.5µs
ZH	**42.35**	38.94	18.07	8.4ms	40.30	**43.53**	17.60	63.8ms	43.76	**51.47**	17.45	109ms
SP	**315.23**	278.83	70.26	35ms	**623.82**	421.29	87.5	8.5ms	**424.36**	354.36	75.20	3.9ms
WL	**109.6**	105.2	26.3	35ms	**223.4**	184.0	31.3	38ms	**310.7**	149.5	22.4	61ms
WPE	0.46µs	**0.34µs**	1.3s	267ms	0.33µs	**0.26µs**	1.8s	620ms	0.26µs	**0.21µs**	1.3s	699ms
WLE	-	**0.44µs**	-	140ms	-	**0.36µs**	-	369ms	-	**0.25µs**	-	144ms
T3W	-	**0.47µs**	-	-	-	**0.37µs**	-	-	-	**0.29µs**	-	-
TM	-	**0.41µs**	-	-	-	**0.31µs**	-	-	-	**0.27µs**	-	-
FS	-	**0.48µs**	-	-	-	**0.34µs**	-	-	-	**0.34µs**	-	-
SPL	-	**112µs**	-	-	-	**260µs**	-	-	-	**596µs**	-	-
FB	-	**241µs**	-	-	-	**353µs**	-	-	-	**729µs**	-	-

although its indexing time and index size are similar to our proposed method, its query time is too time-consuming e.g. more than half a minute to answer a false query on WPE.

Exp 2: synthetic graph performance In our second experiment, we compare our algorithms with LI+ on synthetic datasets. We chose $n = 25,000$ and $L = 8$, and we vary the node degree from 2 to 5 (e.g., number of edges from 10,000 to 25,000), thereby increasing the density of the graph. Our aim here is to understand, using two significantly different synthetic graph models, the impact of graph density on performance, as density is a basic property of graphs. We expect that building indexes on denser graphs will be more difficult for all these methods, as the number of possible paths to explore and index between nodes, and the number of minimal label sets, increase with density. Table 7 summarizes the results. We show the indexing time, index size and average query performance. Note that for the average query performance, we show speed-up of **P2H** and **P2H+** over LI+ and we show the average query time of LI+ in table 7.

What is shown in this table is that **P2H** and **P2H+** can achieve about one order magnitude faster query time with one order magnitude less indexing time and indexing size. Due to the minimal property, indexing size of **P2H+** is always smaller than **P2H**. We can also observe that as D increases, speed-up on average increases. The reason is that the average size of connected components in the graphs increase as D increases and LI+ will perform more work on false queries while **P2H** and **P2H+** will not since they construct the full index.

For NP2H, it shows a similar problem in synthetic graphs. Hence, we do not consider this technique further.

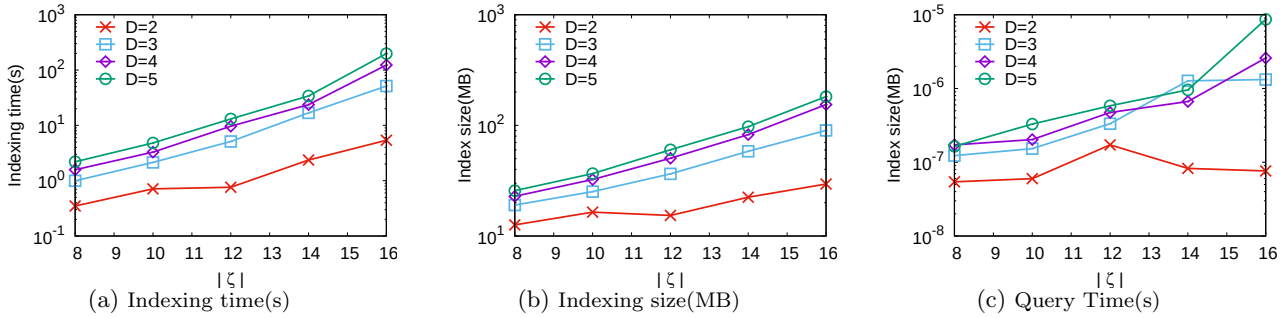
Exp 3: impact of degree and label set size In this experiment, we show the impact of degree and label set size by synthetic graphs. We vary the degree from 2 to 5 and label size from 8,10,12,14 to 16 on ER- and PA-datasets. Figures 4 and 5 show the indexing time, the index size and the query times for the PA-datasets and the ER-datasets. If we are not able to build an index within the 129,600-second (36 hours) time limit, there will be a missing point in the figure. We observe that both the indexing time and the index size grows steadily whenever D is large or small. The growths of the indexing time and index size are stronger for ER-datasets than for PA-datasets. A possible explanation for this might be that ER-datasets have a close to uniform outdegree distribution. On average there are more paths connecting any two vertices, which increases the number of minimal label sets connecting them.

Exp 4: impact of graph structure In our fourth experiment, we analyze the performance of our algorithms with LI+ on PA- and ER-datasets in which we vary the number of vertices n from $5,000$ to $625,000$ with fixed degree $D = 5$. Our goal here is to understand the scalability of these algorithms.

Figures 6 and 7 show the indexing time, the index size and the average query times for PA-datasets and ER-datasets respectively. It is shown that in PA-datasets, the growth of LI+ become faster and faster when n getting larger while in ER-dataset it grows steadily. "As the number of vertices increases, we see that average query times increase for both ER- and PA-datasets, as expected on larger graphs. The PA-datasets exhibit a stronger increase than the ER-datasets. This can be understood as follows. As the ER-datasets have a more uniform out-degree distribution and each vertex has more neighbors, we might have that search on PA-datasets has to explore larger parts of the graph and hence take more time to evaluate a query" since LI+ need to explore all the graph when meet a false query not included in their partial index. For our **P2H** and **P2H+**, the indexing time and index size rise steadily in both PA- and ER-datasets, which shows the superior scalability of our algorithms. As for average query time, **P2H** and **P2H+** can always answer the LCR query within $1us$ when n increasing due to indexing the full LCR information.

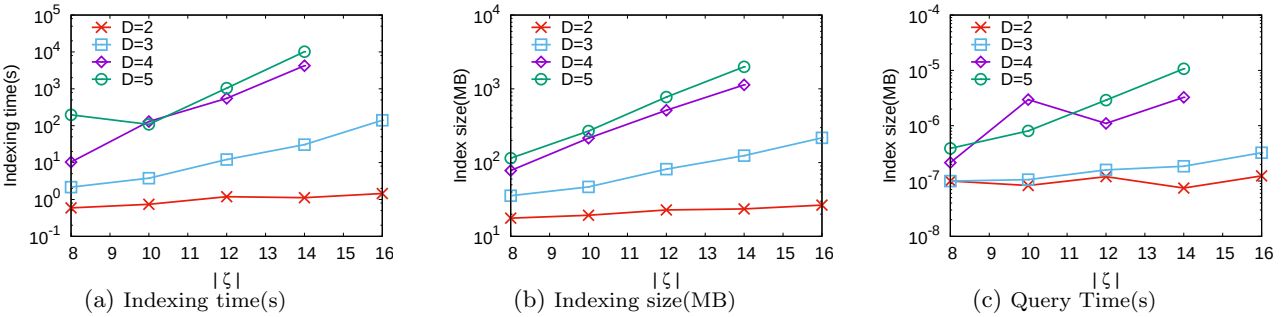
Exp 5: impact of vertex order strategy

In this subsection, we analyze the performance of different node order strategies proposed in this paper. We compare DBS (degree-based strategy) and our ABS (adaptive based strategy). It is shown in Table 8 that query times for these



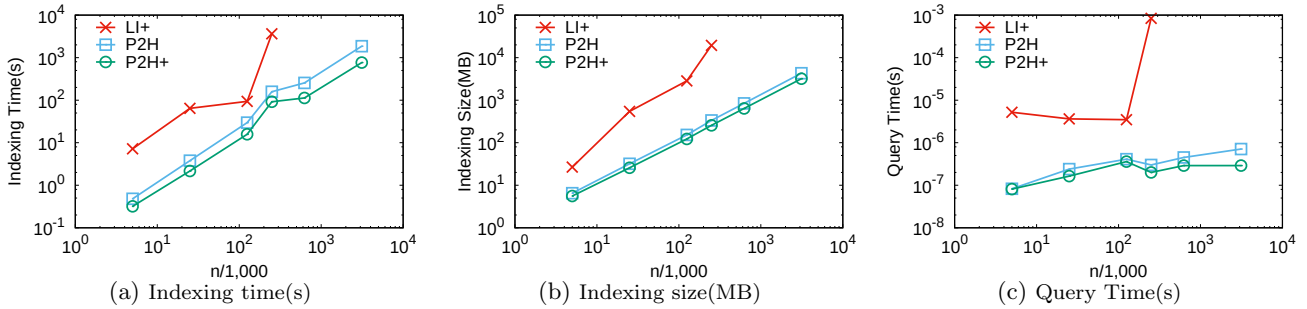
(a) Indexing time(s) (b) Indexing size(MB) (c) Query Time(s)

Figure 4: Indexing time, index size and average query times for PA-datasets with $n = 25,000$ varying the number of vertices. The different lines indicate the node degree (either 2, 3, 4 or 5) of the datasets.



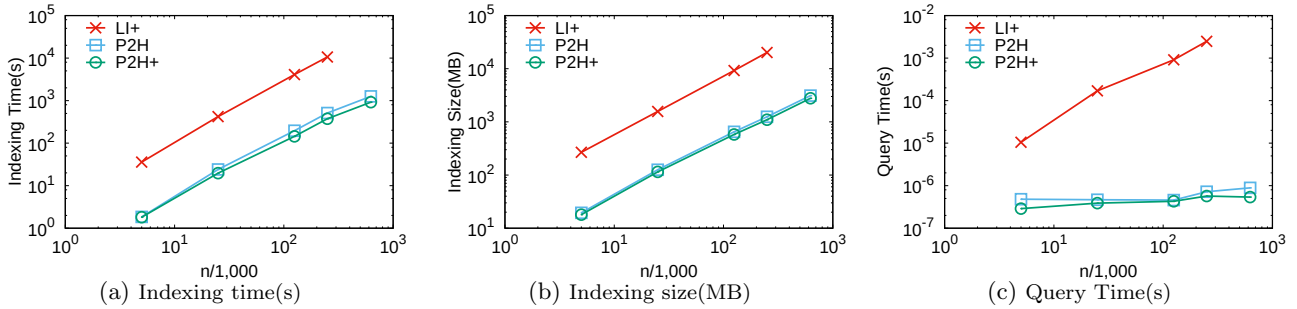
(a) Indexing time(s) (b) Indexing size(MB) (c) Query Time(s)

Figure 5: Indexing time, index size and average query times for ER-datasets with $n = 25,000$ varying the number of vertices. The different lines indicate the node degree (either 2, 3, 4 or 5) of the datasets.



(a) Indexing time(s) (b) Indexing size(MB) (c) Query Time(s)

Figure 6: Indexing time, index size and average query times for PA-datasets with $n = 25,000$ varying the number of vertices. The different lines indicate the node degree (either 2, 3, 4 or 5) of the datasets.



(a) Indexing time(s) (b) Indexing size(MB) (c) Query Time(s)

Figure 7: Indexing time, index size and average query times for ER-datasets with $n = 25,000$ varying the number of vertices. The different lines indicate the node degree (either 2, 3, 4 or 5) of the datasets.

Table 6: Speed-ups of **P2H** and **P2H+** over LI+ for each of the real datasets in false query sets. For LI+, the average query time is given in microseconds. In this table, "-" indicates that the method timed out on this dataset.

Name	$\lvert\zeta/4\rvert$ or 2				$\lvert\zeta/2\rvert$ or 4				$\lvert\zeta-2\rvert$ or 6			
	P2H	**P2H+**	LI+ (µs)	NP2H	**P2H**	**P2H+**	LI+ (µs)	NP2H	**P2H**	**P2H+**	LI+ (µs)	NP2H
RT	1.97	**2.92**	0.134	115µs	1.53	**4.68**	0.128	287µs	1.67	**4.55**	0.1	341µs
ADG	**7.27**	6.59	0.42	1.9ms	5.69	**7.66**	0.34	2.4ms	6.69	**8.51**	0.40	2.3ms
AX	**65**	42.0	32	72ms	**49**	45	28.0	89ms	184	**199**	60.1	110ms
EPIN	**96.73**	83.03	3.41	76ms	136.51	**154.89**	3.22	88ms	**199.73**	173.35	3.67	98ms
SHS	**71.07**	68.34	5.70	8.2ms	130.28	**231.37**	8.05	8.8ms	217.56	**299.52**	8.75	11.9ms
ND	82.82	**110.92**	12.37	21ms	**163.40**	90.72	12.16	43ms	**52.55**	42.8	5.07	25ms
BG	172.65	**223.84**	19.29	30ms	58.48	**183.71**	9.11	34.8ms	80.31	**282.64**	11.75	52.7ms
CT	63.6	**87.1**	20	72ms	**154**	142	35.4	99ms	338	**391**	74	128ms
SFC	**362.42**	309.68	16.60	8.1ms	**632**	598	26.56	7.8ms	**336.58**	325.96	12.39	9.5ms
WS	144.70	**216.27**	20.11	239ms	4,262	**6,987**	677	204ms	9,649	**14,848**	1,301	337ms
WG	**56.44**	53.76	17.02	1.4s	4,336	**6,345**	2,352	748ms	17,091	**21,563**	4,153	1.2s
YT	282.21	**307.51**	326	61ms	194.20	**231**	212	61.8ms	229.44	**246**	139	33.5ms
ZH	195.68	**248.75**	52.1	759ms	1,284	**1,777**	385	713ms	5,586	**6,863**	1,520	766ms
SP	478.49	**522.74**	137.9	4.8s	698	**752**	178	4.7s	**1,769**	1,573	347	4.9s
WL	581.2	**660.5**	63.9	6.1s	1,230	**1,605**	94.7	6.3s	2,277	**2,403**	173.5	6.4s
WPE	0.27µs	**0.18µs**	1.2s	23s	0.22µs	**0.18µs**	1.8s	37s	0.30µs	**0.13µs**	2.3s	37s
WLE	-	**0.26µs**	-	31s	-	**0.25µs**	-	38s	-	**0.22µs**	-	41s
T3W	-	**0.28µs**	-	-	-	**0.34µs**	-	-	-	**0.36µs**	-	-
TM	-	**0.30µs**	-	-	-	**0.34µs**	-	-	-	**0.50µs**	-	-
FS	-	**0.35µs**	-	-	-	**0.31µs**	-	-	-	**0.35µs**	-	-
SPL	-	**91.1µs**	-	-	-	**102µs**	-	-	-	**114µs**	-	-
FB	-	**151µs**	-	-	-	**196µs**	-	-	-	**234µs**	-	-

Table 7: Indexing time(IT) in seconds, index size (IS) in megabytes, and average speed-ups for PA- and ER-datasets with n = 25,000 and L = 8 for which we vary the node degree (from 2 up to 5).

	D	**P2H**		**P2H+**		*LI+*		*NP2H*		*AverageSpeed-up*			
		IT	IS	IT	IS	IT	IS	IT	IS	**P2H**	**P2H+**	LI+(µs)	NP2H(ms)
ER	2	2.3	29.1	0.59	17.7	38.81	624.8	0.32	16.4	176	**239**	24.2	11.9
	3	3.4	43.5	2.2	35.5	29.45	703.4	0.39	16.01	8.04	**11.3**	1.1	13.1
	4	11.4	85.1	10.3	78.1	150.2	1,040	0.48	18.09	155.9	**166.7**	35.8	13.9
	5	24.1	125.8	19.7	114.7	420.2	1,568	0.51	20.42	368.0	**441.0**	172	12.9
PA	2	0.349	13.6	0.337	12.6	4.18	189	0.28	13.0	21.8	**22.9**	1.24	7.5
	3	1.2	21.4	0.997	19.1	20.46	333.5	0.34	15.5	202.9	**242.2**	29.8	2.3
	4	2.26	26.9	1.58	22.9	98	1,160	0.42	18.0	102.9	**130.2**	22.4	4.3
	5	3.79	31.9	2.16	25.8	64.1	538.5	0.53	20.6	14.7	**22.2**	3.65	3.0

Table 8: Indexing time(IT) in seconds, index size(IS) in megabytes and query time(QT) in microsend for real datasets. In this table, "K" indicates 10^3. DBS indicates degree-based strategy and ABS indicates adaptive based strategy.

Name	DBS			ABS		
	IT	IS	QT	IT	IS	QT
RT	**0.007**	0.59	0.04	0.009	**0.53**	**0.036**
ADG	0.125	3.1	0.04	**0.093**	**3.0**	**0.018**
AX	303	404	0.36	**89.4**	**234**	**0.29**
EPIN	9	68	0.05	**7.15**	**63.1**	**0.032**
SHS	4	**37**	0.04	**3.9**	38.0	**0.037**
ND	24	**236**	0.11	**22.1**	247	**0.09**
BG	4	**43**	**0.05**	**3.9**	45	0.07
CT	171	1.1K	**0.19**	**59.6**	**572.8**	0.2
SFC	7	**55**	0.04	**6.5**	56	**0.036**
WS	**50**	**361**	**0.07**	65.4	382	0.073
WG	378	1.2K	0.26	**229**	**1.1K**	**0.2**
YT	1.2K	**365**	0.77	**809**	370	**0.65**
ZH	10.2K	14K	**0.31**	**7.6K**	**11K**	0.32
SP	1.2K	1.6K	0.23	**614**	**1.5K**	**0.18**
WL	**1.9K**	3.8K	**0.13**	2.1K	**3.7K**	0.21
WPE	5.6K	10K	0.22	**3.3K**	**9.2K**	**0.21**
WLE	12.8K	14K	0.29	**11K**	**13K**	**0.25**
T3W	**63.8K**	55K	0.35	77,767	**54K**	**0.29**
TM	79.7K	71.1K	0.36	**77.8K**	**71K**	**0.35**
FS	125K	92.8K	0.36	**109K**	**92K**	**0.33**

two strategies are very similar due to the same framework. For Indexing time and index size, in the case that degrees are sufficient to identify the importance of vertices, DBS and ABS have similar performance and ABS's performance is slightly better. Otherwise, ABS performs significantly better than DBS on graphs AX,CT,WG,ZH and WPE.

7. CONCLUSION

In a directed edge-labeled graph, a fundamental query is to test whether a source vertex can reach a target vertex through a set of edges each of which only contains the given labels. In this paper, we proposed novel label-constrained 2-hop index techniques with some nice properties to efficiently support LCR queries. Our comprehensive experiments showed that our proposed method fully dominated the state-of-the-art technique by means of query time (with up to 5 orders of magnitude speedup), index size and index construction time.

8. ACKNOWLEDGMENTS

Ying Zhang is supported by ARC DP180103096 and FT170100128. Lu Qin is supported by ARC DP160101513. Wenjie Zhang is supported by ARC DP180103096 and DP200101116. Xuemin Lin is supported by NSFC61232006, 2018YFB1003504, ARCDP200101338, ARCDP180103096 and DP170101628.

9. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*, pages 24–35. Springer, 2012.

[2] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 147–154. SIAM, 2014.

[3] T. Akiba, Y. Iwata, and Y. Yoshida. 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*, pages 349–360. ACM, 2013.

[4] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):68, 2017.

[5] P. Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 175–188. ACM, 2013.

[6] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[7] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[8] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen. Distance oracles in edge-labeled graphs. In *EDBT*, pages 547–558, 2014.

[9] M. Chen, Y. Gu, Y. Bao, and G. Yu. Label and distance-constraint reachability queries in uncertain graphs. In *International Conference on Database Systems for Advanced Applications*, pages 188–202. Springer, 2014.

[10] X. Chen, L. Lai, L. Qin, and X. Lin. Structsim: Querying structural node similarity at billion scale. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020.

[11] Y. Chen, Y. Fang, R. Cheng, Y. Li, X. Chen, and J. Zhang. Exploring communities in large profiled graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[12] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 193–204. ACM, 2013.

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

[14] J. Cheng, J. X. Yu, X. Lin, H. Wang, and S. Y. Philip. Fast computation of reachability labeling for large graphs. In *International Conference on Extending Database Technology*, pages 961–979. Springer, 2006.

[15] P. Choudhary. A survey on social network analysis for counterterrorism. *International Journal of Computer Applications*, 112(09), 2015.

[16] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[17] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu. Effective and efficient attributed community search. *The VLDB Journal*, 26(6):803–828, 2017.

[18] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu. Effective community search over large spatial graphs. *PVLDB*, 10(6):709–720, 2017.

[19] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *PVLDB*, 9(12):1233–1244, 2016.

[20] Y. Fang, R. Cheng, S. Luo, J. Hu, and K. Huang. C-explorer: browsing communities in large graphs. *PVLDB*, 10(12):1885–1888, 2017.

[21] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *The VLDB Journal*, 2019.

[22] Y. Fang, Z. Wang, R. Cheng, X. Li, S. Luo, J. Hu, and X. Chen. On spatial-aware community search. *TKDE*, 31(4):783–798, 2019.

[23] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu. Effective and efficient community search over large directed graphs. *TKDE*, 31(11):2093–2107, 2019.

[24] Y. Fang, Y. Yang, W. Zhang, X. Lin, and X. Cao. Effective and efficient community search over large heterogeneous information networks. *PVLDB*, 13(6), Feb. 2020.

[25] Y. Fang, K. Yu, R. Cheng, L. V. S. Lakshmanan, and X. Lin. Efficient algorithms for densest subgraph discovery. *PVLDB*, 12(11):1719–1732, July 2019.

[26] M. S. Hassan, W. G. Aref, and A. M. Aly. Graph indexing for shortest-path finding over dynamic sub-graphs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1183–1197. ACM, 2016.

[27] J. Hu, R. Cheng, K. C.-C. Chang, A. Sankar, Y. Fang, and B. Y. Lam. Discovering maximal motif cliques in large heterogeneous information networks. In *International Conference on Data Engineering (ICDE)*, pages 746–757. IEEE, 2019.

[28] J. Hu, X. Wu, R. Cheng, S. Luo, and Y. Fang. On minimal steiner maximum-connected subgraph queries. *TKDE*, 29(11):2455–2469, 2017.

[29] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 123–134. ACM, 2010.

[30] R. Jin, N. Ruan, S. Dey, and J. Y. Xu. Scarab: scaling reachability computation on large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 169–180. ACM, 2012.

[31] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *Proceedings of the VLDB Endowment*, 6(14):1978–1989, 2013.

[32] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 813–826. ACM, 2009.

[33] P. Klodt, G. Weikum, S. Bedathur, and S. Seufert. Indexing strategies for constrained shortest paths over large social networks. *Universitat des Saarlandes*, 2011.

[34] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[35] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, Y. Zhang, Z. Qian, and J. Zhou. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.*, 12(10):1099–1112, June 2019.

[36] J. Leskovec. Snap: Stanford large network dataset collection, 2016.

[37] Y. Li, M. L. Yiu, N. M. Kou, et al. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11(4):445–457, 2017.

[38] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*, pages 1130–1141, 2019.

[39] B. Liu, F. Zhang, C. Zhang, W. Zhang, and X. Lin. Corecube: Core decomposition in multilayer graphs. In *International Conference on Web Information Systems Engineering*, pages 694–710. Springer, 2019.

[40] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *International Conference on Extending Database Technology*, pages 237–255. Springer, 2004.

[41] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1009–1020. IEEE, 2013.

[42] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1-3):325–346, 1988.

[43] L. D. Valstar, G. H. Fletcher, and Y. Yoshida. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 345–358. ACM, 2017.

[44] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2016.

[45] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 913–924. ACM, 2011.

[46] S. Wadhwa, A. Prasad, S. Ranu, A. Bagchi, and S. Bedathur. Efficiently answering regular simple path queries on large labeled networks. *Age*, 3:v7, 2019.

[47] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin. Efficient computing of radius-bounded k-cores. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 233–244. IEEE, 2018.

[48] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang. Vertex priority based butterfly counting for large-scale bipartite networks. *Proceedings of the VLDB Endowment*, 12(10):1139–1152, 2019.

[49] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang. Efficient bitruss decomposition for large-scale bipartite graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020.

[50] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *Proceedings of the VLDB Endowment*, 7(12):1191–1202, 2014.

[51] P. T. Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, 2012.

[52] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1601–1606. ACM, 2013.

[53] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.

[54] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. Springer, 2010.

[55] Y. Yuan, X. Lian, G. Wang, Y. Ma, and Y. Wang. Constrained shortest path query in a large time-dependent graph. *Proceedings of the VLDB Endowment*, 12(10):1058–1070, 2019.

[56] F. Zhang, C. Li, Y. Zhang, L. Qin, and W. Zhang. Finding critical users in social communities: The collapsed core and truss problems. *IEEE Transactions on Knowledge and Data Engineering*, 32(1):78–91, 2018.

[57] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin. Efficiently reinforcing social networks over user engagement and tie strength. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 557–568. IEEE, 2018.

[58] X. Zhang and M. T. Özsu. Correlation constraint shortest path over large multi-relation graphs. *Proceedings of the VLDB Endowment*, 12(5):488–501, 2019.

[59] Z. Zhang, J. X. Yu, L. Qin, Q. Zhu, and X. Zhou. I/o cost minimization: reachability queries processing over massive graphs. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 468–479. ACM, 2012.

[60] L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems*, 40:47–66, 2014.