DBL: Reachability Queries on Dynamic Graphs

Technical Report

Qiuyi Lyu, Yuchen Li, Bingsheng He, Bin Gong

January 4, 2019

Contents

Abstract

Reachability query is a fundamental problem on graphs that has been extensively studied in the academia and the industry. Many techniques focus on building the index on *static* graphs to achieve query processing in real-time. In many applications, graphs are subject to frequent updates (e.g., social media constantly establishes new followers and inserts edges into the underlying social graphs). Although several update-friendly indexes for reachability query have been proposed, these solutions heavily rely on maintaining the Directed Acyclic Graph (DAG) to facilitate efficient query processing. However, DAG maintenance is a substantial overhead which fails to support the high frequency of graph update. In this paper, we propose the DBL framework which, to the best of our knowledge, is the first DAG-free index to support reachability query on dynamic graph. DBL builds on two complementary indexes: Dynamic Landmark (DL) label and Bidirectional Leaf (BL) label. The former leverages landmark nodes to quickly determine reachable pairs whereas the later prunes unreachable pairs by indexing the leaf nodes in the graph. In addition, DBL is designed to be efficiently implemented under the widely available parallel architectures. We evaluate DBL against the state-of-the-art approaches on dynamic reachability index with extensive experiments on real-world datasets. The results have demonstrated that our sequential implementation achieves orders of magnitude speedup in terms of index update, while still producing competitive query efficiency. On top of that, the multi-cores as well as GPU implementations of DBL show a significant performance boost compared with the sequential implementation.

1 Introduction

Given a graph G and a pair of vertices u and v, reachability query (denoted as $q(u, v)$), a fundamental graph operation answers whether there exists a path from u to v on G. This operation is a core component in supporting numerous applications in reality, ranging from social networks, biological complexes, knowledge graphs, transportation networks and etc. To push the limit of spontaneous query processing, a plethora of index-based approaches have been developed over a decade [29, 7, 21, 27, 32, 22, 24, 34] and demonstrated great success in handling reachability query on graphs with millions of vertices and edges. Nonetheless, the benefits of index do not come for free. Updating the index often incurs an expensive processing cost. Existing indexes designed for *static* graphs are thus not suitable for the dynamic scenario. In many cases, graphs are highly *dynamic*: thousands of new friendships form on social networks like Facebook and Twitter, knowledge graphs are constantly updated with new entities and relations, and transportation networks are subject to changes when road constructions occur.

There have been some efforts in developing reachability index to support graph updates [3, 8, 10, 12, 19, 20, 21]. However, there is a major assumption made in recent works: the Strongly Connected Components (SCCs) in the underlying graph remain unchanged after the graph gets updated. The Directed Acyclic Graph (DAG) collapses SCCs into vertices and the reachability query is then processed on a much smaller graph than the original. The state-of-the-art solutions [35, 29] thus rely on DAG to design index for efficient query processing, yet their index maintenance mechanisms only support the update which does not trigger SCC merge/split in DAG. In practice, such an assumption is *not* valid as edge insertions/deletions could lead to updates of SCCs in DAG. When those happen, one needs to carefully restore DAG to fulfill the prerequisite of existing index update mechanisms. Unfortunately, DAG maintenance is a costly process that cannot be ignored [34].

Motivated by the drawbacks of existing works, we propose a DAG-free dynamic reachability framework (DBL) that supports fast query processing and enables efficient index update at the same time on large scale graphs. DBL is built on the top of two novel index components: (1) a Dynamic Landmark (DL) label, and (2) a Bidirectional Leaf (BL) label. The benefit of combining DL and BL in the DBL is twofold:

- DL can quickly determine reachable pairs while BL, which complements DL, prunes disconnected pairs to remedy the ones that can not be immediately determined by DL. The fusion of DL and BL makes the DBL framework suitable for processing graphs with different characteristics on connectivity.

- Both DL and BL labels are lightweight indexes where each vertex only stores a constant size label. Efficiently pruned Breadth First Search (BFS) is employed to update DL and BL against frequent graph updates.

DL is inspired by the 2-hop index approach [7]. The 2-hop label of a node u consists of $L_{out}(u)$ and $L_{in}(u)$, where $L_{out}(u)$ is a subset of nodes that u can reach and $L_{in}(u)$ is a subset of nodes that can reach u. The 2-hop index guarantees that u reaches v if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. However, adapting the 2-hop label to the dynamic setting leads to an explosion of the index size towards storing the complete transitive closure (TC), which renders its infeasibility against a stream of updates. To mitigate index explosion, the proposed DL label maintains a small set of landmark nodes as the label for each vertex in the graph. Given a graph G with n vertices and k landmark nodes, DL requires $\Theta(kn)$ space in the worst cases. With DL and a query $q(u, v)$, we can quickly determine v is reachable by u when a path node connecting them belong to the 2-hop cover of landmark nodes. We otherwise invoke pruned BFS to process $q(u, v)$ in case DL cannot confirm the reachability.

It is noted that DL could degrade to plain BFS search when the underlying graph is poorly connected, as most reachability queries cannot be answered by DL label. We thus devise BL label to quickly prune vertex pairs that are not reachable to limit the number of costly BFS. BL complements DL and it focuses on building labels around the *"leaf"* nodes in the graph. The leaf nodes in this paper are referred to be the ones which have low centrality in the graph, and they form an exclusive set apart from the landmark node set. For example, the vertices are selected as the leaf nodes if their in-degree or out-degree is zero. BL label of a vertex u is defined to be the leaf nodes which can either reach u or u can reach them. With BL, a vertex u does not reach v if the label containment condition is not satisfied. Even when BL cannot give an affirmative answer to $q(u, v)$ immediately, it offers effective pruning while we perform BFS from u to v and the number of traversed vertices is drastically reduced.

Aside from the query processing power of DL and BL label, we note that DL and BL are lightweight indexes which are manipulated with simple and compact bitwise operations. Both labels do not require maintaining DAG and are updated with efficiently pruned BFS traversal. It thus enables efficient deployment on the widely available parallel architectures. Given many parallel programming interfaces like OpenMP and CUDA, the DBL framework can be deployed to various platforms with ease.

Hereby, we summarize the contributions as the following:

- We introduce the DBL framework which combines two complementary DL and BL labels to enable efficient reachability query processing on large graphs.
- We propose novel index update algorithms for DL and BL. To the best of our knowledge, this is the first solution for dynamic reachability index without maintaining a DAG. In addition, the algorithms can be easily implemented with parallel interfaces.
- We conduct extensive experiments to validate the performance of DBL against the state-of-the-art dynamic methods. DBL achieves competitive query performance and up to three orders of magnitudes speedup for index update. We also implement DBL on multi-cores and GPU-enabled system and demonstrate significant performance boost compared with our sequential implementation.

The remaining part of this paper is organized as follows. Section 2 presents the related works. The notations as well as the DBL framework are introduced in Section 3. Sections 4 and 5 demonstrate the designs of DL and BL labels respectively. Section 6 reports the experimental results. Finally, we conclude the paper in Section 7.

2 Related Work

Reachability query is a fundamental graph operation that has been extensively studied over decades. Other than direct graph traversal, e.g., BFS and DFS, the existing works can be broadly classified into two categories: *Label-Only* approach and *Label+G* approach.

Label-Only approach tries to compress the entire Transitive Closure (TC) for label construction and process queries by examining the label [35, 4, 11, 5, 13, 6, 31, 7, 14, 21, 9, 27, 1]. There are in general

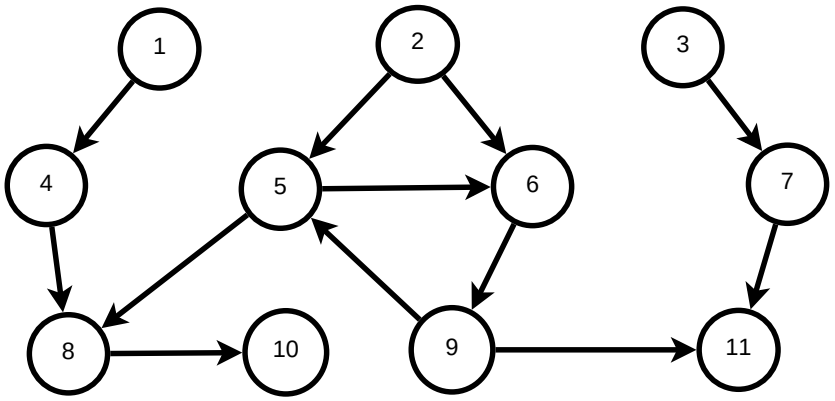


Figure 1: A running example of graph G

two methods for *Label-Only* approach: the 2-hop label and the interval label. The 2-hop label is first proposed in [7]. The method stores a set of nodes as the label for each vertex in the graph and it guarantees that u reaches v iff the labels of u and v have non-zero overlap. The interval label, which is firstly introduced by [1], uses pre-post (or min-post) labeling on a spanning tree of `DAG`. Pre-post label assigns $L = [s, e]$ to each vertex in the graph where s is the vertex's pre-order rank and e is the vertex's post-order rank with respect to `DFS`. For vertices $u, v \in V$, if $s_u < s_v$ and $e_u > e_v$, it then concludes that u could reach v. There are also a number of solutions proposed to extend the 2-hop or the interval labels to further improve the performance in terms of label size, query processing and label construction efficiency [9, 27, 14, 4, 15].

Label+G approach is very popular in recent years [24, 25, 32, 33, 29, 28, 26, 22]. The labels constructed in *Label+G* approach *partially* summarize the reachability information in `TC`. Thus, a pruned graph traversal will be invoked to process a reachability query if the labels cannot give a direct answer. Most of the existing *Label+G* based solutions [25, 32, 33, 22] are built on top of the interval label. Compared with *Label-Only* approach, solutions [33, 22] for *Label+G* are shown to achieve a better balance between the index size and the query processing efficiency empirically.

The aforementioned solutions for reachability query are mostly designed for static graphs, which means the index (e.g., the labels) needs to be rebuilt once the graph is updated. There have been some studies on dynamic graph [3, 8, 10, 12, 19, 20, 21]. Yildirim et al. propose `DAGGER` [34] which maintains the graph as a `DAG` after insertions and deletions. The index is constructed on `DAG` to facilitate reachability query processing. The main operation for `DAG` maintenance is the merge and split of *Strongly Connected Component* (`SCC`). Unfortunately, it has been shown that `DAGGER` exhibits unsatisfactory performance on handling large graphs with millions of vertices [35]. The state-of-the-art approaches: `TOL` [35] and `IP` [29] follow the maintenance method for `DAG` of `DAGGER` and propose novel dynamic index on `DAG` to improve the query processing performance. However, the overheads of `SCC` maintenance are excluded in their experiments [35, 29] and such overheads is in fact non-negligible [34]. Moreover, as shown in our experiments, the update overheads of `TOL` and `IP` dominates the overall processing workload on dynamic graphs, even when `DAG` maintenance overhead is ignored [35, 29]. In this paper, we propose the `DBL` framework which only maintains the labels for all vertices in the graph without constructing `DAG`. `DBL` achieves competitive query processing performance with the state-of-the-art solutions (i.e., `TOL` and `IP`) while offering orders of magnitude speedup in terms of index update.

3 DBL overview

In this section, we will present some basic notations in Section 3.1 and then introduce the definitions of `DL` and `BL` in Section 3.2. Finally, the `DBL` processing framework is proposed in Section 3.3.

v	$\texttt{DL}_{in}$	$\texttt{DL}_{out}$
1	$\emptyset$	8
2	$\emptyset$	5,8
3	$\emptyset$	$\emptyset$
4	$\emptyset$	8
5	5	5,8
6	5	5,8
7	$\emptyset$	$\emptyset$
8	5,8	8
9	5	5,8
10	5,8	$\emptyset$
11	5	$\emptyset$

Figure 2: DL label for G

v	$\texttt{BL}_{in}$	$\texttt{BL}_{out}$
1	1	10
2	2	10,11
3	3	11
4	1	10
5	2	10,11
6	2	10,11
7	1,3	11
8	1,2	10
9	1,2	10,11
10	1,2	10
11	1,2,3	11

Figure 3: BL label for G

3.1 Notations

A directed graph is defined as $G = (V, E)$, V is the vertex set and E is the edge set with $n = |V|$ and $m = |E|$. We denote the reversed graph of G as $G' = (V, E')$ where all the edges of G are in the opposite direction in G'. In this paper, the forward direction refers to traversing on the edges in G. Symmetrically, the backward direction refers to traversing on the edges in G'. We denote an edge from vertex u to vertex v as (u, v). A path from u to v in G is denoted as $Path(u, v) = (u, w_1, w_2, w_3, \ldots, v)$ where $w_i \in V$ and the adjacent vertices on the path are connected by an edge in G. We say v is reachable by u iff there exists a $Path(u, v)$. In addition, we use $Suc(u)$ to denote the direct successors of u and the direct predecessors of u are denoted as $Pre(u)$. Similarly, we denote all the ancestors of u (including u) as $Anc(u)$ and all the descendants of u (including u) as $Des(u)$.

We denote $q(u, v)$ as a reachability query from u to v. In this paper, we study the dynamic scenario where vertices/edges can be inserted and deleted from the graph. We focus on presenting edge insertion/deletion and vertex insertion/deletion is discussed in Section 5.2.

Example 1. *Figure 1 is the graph that we use as a running example throughout the presentation of this paper. For vertex 9,* $Anc(9) = \{5, 6, 9, 2\}$ *and* $Des(9) = \{5, 6, 8, 9, 10, 11\}$*. For query* $q(7, 10)$*, the answer will be false as there is no path from vertex 7 to vertex 10. After inserting an edge from vertex 7 to vertex 4, a new path forms* $Path(7, 10) = (7, 4, 8, 10)$ *and* $q(7, 10)$ *returns true after the edge insertion. Similarly, vertex 2 could reach vertex 11 in the graph as* $Path(2, 11) = (2, 6, 9, 11)$*. If we delete the edge* $(6, 9)$*, vertex 11 will not be reachable by vertex 2.*

3.2 DBL Definitions

The DBL framework consists of two index components to handle dynamic reachability query, which are defined as the following.

Definition 1 (DL label)**.** *Given a vertex set denoted as the landmark set* $L \subset V$ *and* $|L| = k$*, we define two labels for each vertex* $v \in V$*:* $\texttt{DL}_{in}(v)$ *and* $\texttt{DL}_{out}(v)$*.* $\texttt{DL}_{in}(v)$ *is a subset of nodes in* L *that could reach* v *and* $\texttt{DL}_{out}(v)$ *is a subset of nodes in* L *that* v *could reach.*

In other words, if there exists a vertex $l \in L$ such that l is an intermediate vertex on $Path(u, v)$, then l appears in $\texttt{DL}_{out}(u)$ and $\texttt{DL}_{in}(v)$. It is noted that DL label is a subset of the 2-hop label [7]. The following lemma shows an important property of DL label for reachability query processing.

Corollary 1. *Given two vertices u,v and their corresponding* DL *label,* $\texttt{DL}_{out}(u) \cap \texttt{DL}_{in}(v) \neq \emptyset$ *deduces u reaches v but not vice versa.*

Example 2. *We continue the running example in Figure 1. Assuming the landmark set is chosen as* $\{5, 8\}$*, the corresponding* DL *label is shown in Figure 2.* $q(1, 10)$ *returns true since* $\texttt{DL}_{out}(1) \cap \texttt{DL}_{in}(10) = \{8\}$*. However, the labels cannot give negative answer to* $q(3, 11)$ *despite* $\texttt{DL}_{out}(3) \cap \texttt{DL}_{in}(11) = \emptyset$*. This is because the intermediate vertex* 7 *on the path from* 3 *to* 11 *is not included in the landmark set.*

To achieve good performance, we need to select a set of vertices as the landmarks such that they cover most of the reachable vertex pairs in the graph, i.e., $\texttt{DL}_{out}(u) \cap \texttt{DL}_{in}(v)$ contains at least one landmark node for any reachable vertex pair u and v. The optimal landmark selection has been proved to be NP-hard [18]. In this paper, we adopt a heuristic method for selecting DL label nodes following existing works [2, 18]. In particular, we rank vertices with $M(u) = |Pre(u)| \cdot |Suc(u)|$ to approximate their centrality and select top-k vertices with the largest $M(u)$ as DL label nodes.

DL label is selected according to *high centrality* for covering as much reachability information as possible. However, DL label is efficient for giving positive answer to a reachability query whereas BL label plays a complementary role by pruning unreachable pairs. On a high level, we wish to select vertices for BL with *low centrality* to complement DL. Since low-centrality nodes refer to those who hardly connect other vertices in the graph, choosing these nodes for BL label help to distinguish different vertices in terms of how they reach BL label nodes and quickly determine if two vertices are *not* reachable.

Definition 2 (BL label)**.** *We also introduce two labels for each vertex $v \in V$: $\texttt{BL}_{in}(v)$ and $\texttt{BL}_{out}(v)$. $\texttt{BL}_{in}(v)$ contains all the zero in-degrees vertices that can reach v, and $\texttt{BL}_{out}(v)$ contains all the zero out-degrees vertices that could be reached by v. For convenience, we refer the vertices with either zero in-degree or out-degree as the leaf nodes.*

Corollary 2. *Given two vertices u,v and their corresponding* BL *label, u does not reach v in G if $\texttt{BL}_{out}(v) \not\subseteq \texttt{BL}_{out}(u)$ or $\texttt{BL}_{in}(u) \not\subseteq \texttt{BL}_{in}(v)$.*

BL label can give a negative answer to $q(u, v)$. This is because if u could reach v, then u could reach all the leaf nodes that v could reach, and all the leaf nodes that reach u should also reach v. Thus, to ensure the pruning power, we need to carefully select leaf nodes to differentiate u,v in terms of their reachability information to these leaf nodes. It is noted that the general graph does not necessarily contain vertices of zero degree. Our approach can also incorporate *low* degree vertices as the leaf nodes. To simplify the presentation, we focus on the leaf nodes in the main body of this paper and leave additional discussions in the appendix (Section 8.3).

Example 3. *Figure 3 shows* BL *label for the running example.* BL *label gives negative answer to $q(4, 6)$ since $\texttt{BL}_{in}(4)$ is not contained by $\texttt{BL}_{in}(6)$. Intuitively, vertex 1 reaches vertex 4 but cannot reach 6 which indicates 4 should not reach 6.* BL *label cannot give positive answer. Take $q(5, 2)$ for an example, the labels satisfy the containment condition but a positive answer cannot be given.*

To develop efficient index operations, we build a hash set of size k' for BL as the following. Both $\texttt{BL}_{in}$ and $\texttt{BL}_{out}$ are a subset of $\{1, 2, 3, 4, \ldots, k'\}$ where k' is a user-defined label length, and they are stored in bit vectors. A hash function is used to map the leaf nodes to a corresponding bit in BL label. From now on, we use $\texttt{BL}_{in}$ and $\texttt{BL}_{out}$ to denote the hash sets of the corresponding labels. It is noted that one can still use Corollary 2 to prune unreachable pairs with the hashed BL label.

For dynamic updates, we do not need to perform immediate update on the indexes upon every edge update. Although the degree of a vertex may vary, keeping the original label nodes will *not* result in false answer produced for query processing. Besides, landmark nodes with a high degree will not change frequently. In addition, there are many leaf nodes and adding a few leaf nodes does not improve the query efficiency significantly. For label node deletion, we simply handle it as a non-label vertex deletion which we discuss in Section 5.2. The index will be periodically rebuilt to choose a new set of label nodes for maintaining the index pruning power.

Space complexity. The space complexities of DL and BL labels are $O(kn)$ and $O(k'n)$, respectively. As we focus on the dynamic scenario, we leave the index construction for DL and BL in the appendix.

Before moving on, all frequently used notations are summarized in Table 1.

3.3 The DBL Framework

The DBL framework for query on dynamic graph is presented in Figure 4. DBL is built on top of the dynamic graph storage layer. DL and BL label have their independent query and update components. Following existing works on reachability for dynamic graphs [34, 35, 29], DBL assumes a synchronized

Table 1: Frequently used notations

Notation	Descriptions
$G(V, E)$	the vertex set V and the edge set E of a directed graph G
n	the number of nodes in G
m	the number of edges in G
$Pre(u)$	the set of u's in-neighbors
$Suc(u)$	the set of u's out-neighbors
$Des(u)$	the set of u's descendants including u
$Anc(u)$	the set of u's ancestors including u
$Path(u, v)$	A path from vertex u to vertex v
$q(u, v)$	the reachability query from u to v
k	the length of DL label
k'	the length of BL label
$\texttt{DL}_{in}(u)$	the label that keeps all the landmark nodes that could reach u
$\texttt{DL}_{out}(u)$	the label that keeps all the landmark nodes that could be reached by u
$\texttt{BL}_{in}(u)$	the label that keeps all the hash value of the BL label nodes that could reach u
$\texttt{BL}_{out}(u)$	the label that keeps all the hash value of the BL label nodes that could be reached by u

update model where the queries are processed after the index updates are completed. In the remaining part of this section, we first present the query processing mechanism and leave the index update of DL and BL in Section 4 and Section 5 respectively.

Algorithm 1 illustrates the query processing framework of DBL. DBL can be categorized as a *Label+G* approach where BFS is invoked if the labels are unable to answer the query. Given a reachability query $q(u, v)$, we immediately return the answer if the labels are sufficient to determine the reachability (lines 12-19). Otherwise, we turn to BFS search with efficient pruning. Upon visiting a vertex w, the procedure will determine whether the vertex w should be enqueued in lines 27 and 29. BL and DL labels will judge whether the destination vertex v will be in the $Des(w)$. If not, w will be pruned from BFS. We use four pruning conditions depicted in lines 12-19 of Algorithm 1 to quickly answer the query before traversing the graph with BFS. The reachability query will first be answered by DL label from lines 12 to 17. DL gives a positive answer if the label intersection is not empty (line 12-13). There are two early termination rules implemented in lines 14 and 16 respectively. We prove their correctness in Theorem 1 and Theorem 2 respectively.

Theorem 1. *In Algorithm 1, when* DL_*Intersec(x,y) returns false and* DL_*Intersec(y,x) returns true, vertex y is not reachable by vertex x.*

Proof. DL_*Intersec*(y, x) returns true indicates that vertex x is reachable by vertex y. If vertex y is reachable by vertex x, then y and x must be in the same SCC (according to the definition of SCC). As all the vertices in SCC are reachable to each other, the landmark nodes in $\texttt{DL}_{out}(y) \cap \texttt{DL}_{in}(x)$ should also be included in $\texttt{DL}_{out}$ and $\texttt{DL}_{in}$ label for all vertices in the same SCC. This means DL_*Intersec*(x, y) should return true. Therefore x cannot reach y otherwise it contradicts with the fact that DL_*Intersec*(x, y) returns false. □

Theorem 2. *In Algorithm 1, if* DL_*Intersec(x, y) returns false and* DL_*Intersec(x,x) or* DL_*Intersec(y,y) returns true then vertex y is not reachable by vertex x.*

Proof. If DL_*Intersec*(x,x) returns true, it means that vertex x is a landmark or x is in the same SCC with a landmark. If x is in the same SCC with landmark l, according to the definition of SCC, vertex

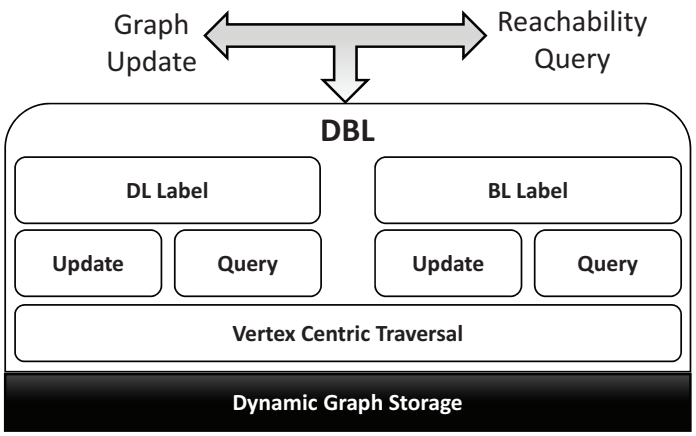


Figure 4: The DBL framework

x and vertex l can be treated as the same vertex in terms of reachability. As landmark l will push its label element l to $\texttt{DL}_{out}$ label for all the vertices in $Anc(l)$ and to $\texttt{DL}_{in}$ label for all the vertices in $Des(l)$. The reachability information for landmark l will be fully covered. It means that x's reachability information is also fully covered. Thus DL label is enough to answer the query without BFS. Hence y is not reachable by x if DL_$Intersec(x, y)$ returns false. The proving process is similar for the case when DL_$Intersec(y,y)$ returns true. □

Query complexity. Given a query $q(u, v)$, the time complexity is $O(k+k')$ when the query can be directly answered by DL and BL labels. Otherwise, we turn to BFS search, which has a worst case time complexity of $O(m + n)$. Let ρ denote the ratio of vertex pairs whose reachability could be directly answered by the label. The amortized time complexity is $O((1-\rho)\cdot(m+n)+\rho\cdot(k+k'))$. Empirically, ρ is over 95% according to our experiments (Table 7) which implies efficient query processing.

It is noted that DL and BL labels complement each other in two different ways for query processing. First, DL quickly determines positive answer which works efficiently for graphs with high connectivity, whereas BL produces negative answer so that expensive BFS search is avoided for graphs with low connectivity. Second, DL tends to select landmark vertices which are parts of large SCC. On the contrary, BL chooses leaf vertices to index BL label, where each leaf vertex forms a SCC with only one vertex. The complementary choices of indexing vertices for DL and BL labels make the framework as whole covers as much reachability information as possible. Hence, the design of the DBL framework avoids the drawbacks of maintaining DAG in existing works. In addition, unlike the previous approaches [29, 33, 24] relying on DAG that heavily use DFS search for query processing, the pruned BFS in DBL naturally follows the BSP (Bulk Synchronize Processing) [17, 23, 30] model, which can be easily implemented with the vertex-centric paradigm. Under the vertex-centric paradigm, the frontier expansion of BFS is parallelized in a synchronized manner.

4 Dynamic Landmark Label

This section presents updates on the DL label.

4.1 DL Insertion Algorithm

When inserting a new edge (u, v), all vertices in $Anc(u)$ should reach all vertices in $Des(v)$. On a high level, all landmark nodes that could reach u should also reach vertices in $Des(v)$. In other words, all the landmark nodes that could be reached by v should also be reached by vertices in $Anc(u)$. Thus, we could either: 1) update the label by adding $\texttt{DL}_{in}(u)$ into $\texttt{DL}_{in}(x)$ for all $x \in Des(v)$; 2) adding $\texttt{DL}_{out}(v)$ into $\texttt{DL}_{out}(x)$ for all $x \in Anc(u)$. In fact, they are identical in terms of maintaining reachability information and only one way needs to be updated. Algorithm 2 depicts the first scenario.

Algorithm 1 Query Processing Framework for DBL

Input: Graph $G(V, E)$, DL label, BL label, $q(u, v)$
Output: Answer of the query.

```
1: function DL_Intersec(x,y)
2:     if DL_out(x) ∩ DL_in(y) then
3:         return true
4:     else
5:         return false
6: function BL_Contain(x,y)
7:     if BL_in(x) ⊆ BL_in(y) and BL_out(y) ⊆ BL_out(x) then
8:         return true
9:     else
10:        return false
11: procedure QUERY(u,v)
12:    if DL_Intersec(u,v) then
13:        return true
14:    if DL_Intersec(v,u) then
15:        return false
16:    if DL_Intersec(u,u) or DL_Intersec(v,v) then
17:        return false
18:    if not BL_Contain(u,v) then
19:        return false
20:    //Query by pruned BFS
21:    Enqueue u for BFS
22:    while queue not empty do
23:        w ← pop queue
24:        for vertex x ∈ Suc(w) do
25:            if x = v then
26:                return true
27:            if DL_Intersec(u,x) then
28:                continue
29:            if not BL_Contain(x,v) then
30:                continue
31:            Enqueue x
32:    return false
```

We test whether the newly inserted edge will add extra connectivity to the graph in line 1. If DL label can determine that vertex v is reachable by vertex u in the original graph before the edge insertion, the insertion will not trigger any label update. Lines 2-8 describe a pruned BFS process. For a visited vertex x, we prune x without traversing $Des(x)$ iff $\texttt{DL}_{in}(u) \subseteq \texttt{DL}_{in}(x)$, because all the vertices in $Des(x)$ are deemed to be unaffected as their $\texttt{DL}_{in}$ labels are supersets of $\texttt{DL}_{in}(x)$.

Example 4. *Figure 5 shows an example of* DL *label update for edge insertion.* $\texttt{DL}_{in}$ *label is presented with brackets. Give an edge* $(9, 2)$ *inserted,* $\texttt{DL}_{in}(9)$ *will be copied to* $\texttt{DL}_{in}(2)$ *as* $\texttt{DL}_{in}(2)$ *is an empty set. Then an inspection will be processed on* $\texttt{DL}_{in}(5)$ *and* $\texttt{DL}_{in}(6)$*. Since* $\texttt{DL}_{in}(9)$ *is a subset of* $\texttt{DL}_{in}(5)$ *and* $\texttt{DL}_{in}(6)$*, vertex 5 and vertex 6 are pruned from* BFS *and the update progress is terminated.*

4.2 DL Deletion Algorithm

Deleting an edge is more complicated than inserting an edge. When an edge (u, v) is deleted, not all the connectivities between $Anc(u)$ and $Des(v)$ are affected. The status of a vertex pair's reachability changes *iff* the paths connecting them always include the deleted edge. Thus, we need to carefully adjust the label so that only the reachability information carried by the deleted edge is removed.

Algorithm 2 DL label update for edge insertion

Input: Graph $G(V, E)$, DL label, Inserted edge (u, v)
Output: Updated DL label

```
1: if DL_out(u) ∩ DL_in(v) == ∅ then
2:     Initialize an empty queue and enqueue v
3:     while queue is not empty do
4:         p ← pop queue
5:         for vertex x ∈ Suc(p) do
6:             if DL_in(u) ⊈ DL_in(x) then
7:                 DL_in(x) ← DL_in(x) ∪ DL_in(u)
8:                 enqueue x
```

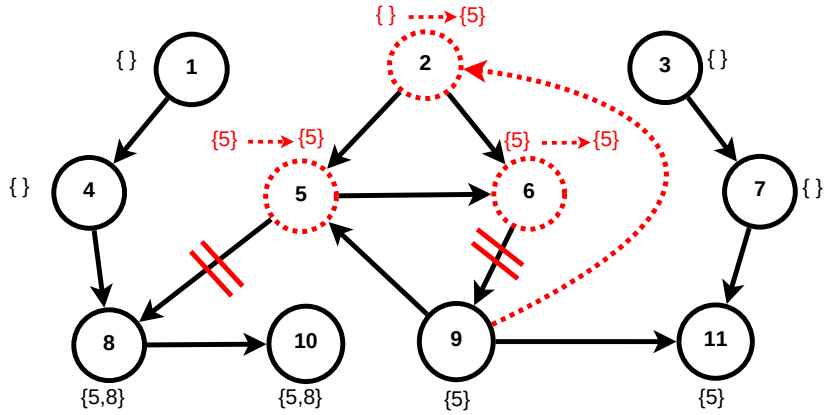


Figure 5: DL label update for inserting edge $(9, 2)$

Our deletion algorithm is based on an important observation that if a vertex u is not a landmark, $\mathtt{DL}_{in}(u)$ and $\mathtt{DL}_{out}(u)$ satisfy the following equation:

$$\begin{aligned} \mathtt{DL}_{in}(u) &= \bigcup_{p \in Pre(u)} \mathtt{DL}_{in}(p) \\ \mathtt{DL}_{out}(u) &= \bigcup_{q \in Suc(u)} \mathtt{DL}_{out}(q) \end{aligned} \tag{1}$$

In other words, all the elements in $\mathtt{DL}_{in}(u)$ are passed from vertices in $Pre(u)$ and all the elements in $\mathtt{DL}_{out}(u)$ label are passed from vertices in $Suc(u)$. This is because when we construct/insert DL label, the label will be propagated in a BFS manner (Algorithm 2). If vertex x is visited in DL label insertion, all the vertices belonging to $Des(x)$ will be visited. As a result, $\mathtt{DL}_{in}$ label of the vertices belong to $Des(x)$ will be a superset of $\mathtt{DL}_{in}(x)$. Similarly, $\mathtt{DL}_{out}$ labels of all vertices in $Anc(x)$ will be supersets of $\mathtt{DL}_{out}(x)$. The only exception is when the vertex u is a landmark, both $\mathtt{DL}_{out}(u)$ and $\mathtt{DL}_{in}(u)$ will always include itself.

To properly update DL label, we need to make sure that the labels that uniquely passed through the deleted edge in the construction process will be excluded from the affected vertices. For deleted edge (u, v), take $\mathtt{DL}_{in}$ label as example, if there exist labels that uniquely pass through (u, v), we need to check the vertices that belong to $Des(v)$. Hence, all the vertices in $Des(v)$ will check $\mathtt{DL}_{in}$ label passed by their predecessors. $\mathtt{DL}_{out}$ label for the vertices in $Anc(u)$ will be handled in a similar manner. In summary, all the vertices belong to $Des(v)$ will check their predecessors' $\mathtt{DL}_{in}$ label to make sure the edge deletion has been reflected in the label. Symmetrically, the vertices belong to $Anc(u)$ will check their successors' $\mathtt{DL}_{out}$ label to guarantee the consistency.

DL label update process for edge deletion is presented in Algorithm 3. We initialize a set $R_{in}(v)$ (line 1), which keeps the elements that should be excluded from $\mathtt{DL}_{in}(v)$. $R_{in}(v)$ is computed by taking

Algorithm 3 DL Label update for edge deletion

Input: Graph $G(V, E)$, DL label, Deleted edge (u, v)
Output: Updated DL label

1: // Update $\texttt{DL}_{in}$ in the forward direction
2: Initialize $R_{in}(v) \leftarrow \texttt{DL}_{in}(u)$
3: **for all** p in *Pre(v)* **do**
4: $\quad R_{in}(v) = R_{in}(v) - \texttt{DL}_{in}(p)$
5: **if** v is landmark vertex **then**
6: $\quad R_{in}(v) = R_{in}(v) - v$
7: **if** $R_{in}(v) \neq \emptyset$ **then**
8: $\quad \texttt{DL}_{in}(v) = \texttt{DL}_{in}(v) - R_{in}(v)$
9: $\quad$ Initialize an empty queue and enqueue $(v, R_{in}(v))$
10: $\quad$ **while** queue is not empty **do**
11: $\quad\quad (p, R_{in}(p)) \leftarrow$ pop queue
12: $\quad\quad$ **for** node x in *Suc(p)* **do**
13: $\quad\quad\quad R_{in}(x) = R_{in}(p)$
14: $\quad\quad\quad$ **for** node y in *Pre(x)* **do**
15: $\quad\quad\quad\quad R_{in}(x) = R_{in}(x) - \texttt{DL}_{in}(y)$
16: $\quad\quad\quad$ **if** x is landmark vertex **then**
17: $\quad\quad\quad\quad R_{in}(x) = R_{in}(x) - x$
18: $\quad\quad\quad$ **if** $R_{in}(x) \neq \emptyset$ **then**
19: $\quad\quad\quad\quad \texttt{DL}_{in}(x) = \texttt{DL}_{in}(x) - R_{in}(x)$
20: $\quad\quad\quad\quad$ enqueue $(x, R_{in}(x))$
21: // Update $\texttt{DL}_{out}$ in the backward direction, which is symmetrical to the update of $\texttt{DL}_{in}$ (Lines 2-20).

the difference between $\texttt{DL}_{in}(v)$ with $\texttt{DL}_{in}$ labels of v's predecessor (Lines 3-4). Subsequently, we will check whether vertex v is a landmark in lines 5-6, because the landmark will always keep its itself as a label element in both $\texttt{DL}_{in}$ and $\texttt{DL}_{out}$. If $R_{in}(v)$ is an empty set, it means no label element will be excluded from $\texttt{DL}_{in}(v)$ and the edge (u, v) does not uniquely carry reachability information. Hence all the vertices in $Des(v)$ will be unaffected and the update process will be terminated. If the $R_{in}(v)$ set is not empty, pruned BFS process will be taken to inspect vertices in $Des(v)$ (lines 9-20). For vertex p visited by BFS, we will inspect all $\texttt{DL}_{in}$ labels of the vertices belong to $Suc(p)$(lines 12-20), compute R_{in} by set difference operations from lines 14 to 15 and check if the vertex is a landmark (lines 16-17). If the R_{in} set is not empty (lines 18-20), the vertex will be enqueued as its descendants's label may be altered by the edge deletion. We also need to update $\texttt{DL}_{out}$ in the backward direction. Since it is symmetrical to the update of $\texttt{DL}_{in}$, we omit the pseudo code.

Example 5. *Figure 6 shows an example of* DL *label update after edge deletion. If edge* $(6, 9)$ *is deleted from the graph, the label element "5" in* $\texttt{DL}_{in}(9)$ *is uniquely passed from vertex 6 as vertex 6 is the only predecessor of vertex 9. Therefore, "5" will be included in* $R_{in}(9)$ *and excluded from* $\texttt{DL}_{in}(9)$*. After the update of* $\texttt{DL}_{in}(9)$*,* BFS *process will inspect the successors of vertex 9. When vertex 11 is visited, it finds that label element "5" will not be passed from edge (7,11). As a result, "5" will be excluded from* $\texttt{DL}_{in}(11)$*. Henceforth, vertex 5 will be inspected. Since vertex 5 is a landmark,* $R_{in}(5)$ *will exclude 5 and become an empty set. Vertex 5 will not be enqueued and the update process is terminated.*

5 Bidirectional Leaf Label

DL label only gives positive answer to a reachability query. In poorly connected graphs, DL will degrade to expensive BFS search. In this section, we present how to update DL label.

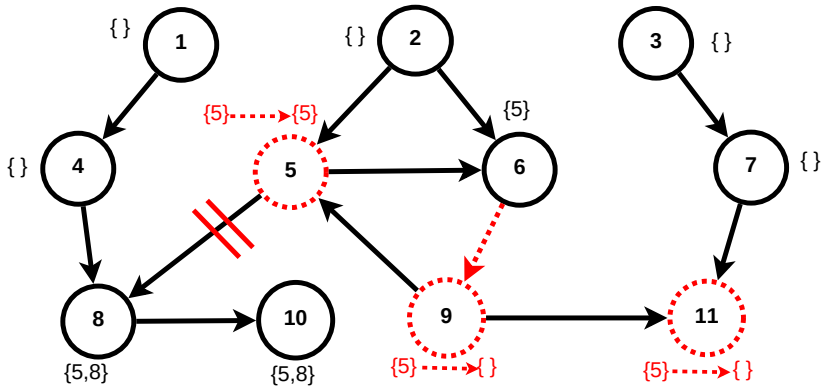


Figure 6: DL label update for deleting edge $(6, 9)$

Algorithm 4 BL label update for edge insertion

Input: Graph $G(V, E)$, BL label, Inserted edge (u, v)
Output: Updated BL label

```
1: if DL_out(u) ∩ DL_in(v) == ∅ then
2:     Initialize an empty queue and enqueue v
3:     while queue is not empty do
4:         p ← pop queue
5:         for vertex x ∈ Suc(p) do
6:             if BL_in(u) ⊈ BL_in(x) then
7:                 BL_in(x) ← BL_in(x) ∪ BL_in(u)
8:                 enqueue x
```

5.1 BL Update Algorithms

For BL label, when inserting an edge $(u, v) \notin E$, we need to guarantee that all the zero in-degree vertices that reach u could reach all the vertices belonging to $Des(v)$. Symmetrically, all the zero out-degree vertices that could be reached by v should now be reachable by vertices in $Anc(u)$. For DL label, one needs to update $\texttt{DL}_{in}$ label in v's descendant direction or update $\texttt{DL}_{out}$ label in u's ancestor direction, whereas BL is required to update labels in both directions. More specifically, we need to maintain the set containment condition: $\texttt{BL}_{in}(x) \subseteq \texttt{BL}_{in}(y)$ and $\texttt{BL}_{out}(y) \subseteq \texttt{BL}_{out}(x)$ where vertex $x \in Anc(u)$ and $y \in Des(v)$.

Due to space limit, we only show the update process for $\texttt{BL}_{in}(\cdot)$ label in Algorithm 4. The update process for $\texttt{BL}_{out}(\cdot)$ is omitted since it is similar to $\texttt{BL}_{in}(\cdot)$ update process. We will first check whether vertex v is reachable by vertex u (line 1) to make sure if the newly insert edge will add extra connectivity or not. Inserted edge with no extra connectivity added will not incur BL label update. Lines 3-9 describe a forward BFS in which we update the $\texttt{BL}_{in}$ label for vertices in $Des(v)$. We will prune vertex x if the $\texttt{BL}_{in}$ label for vertices in $Des(x)$ will not be affected after the edge insertion. For a visited vertex x, "not affected" means that $\texttt{BL}_{in}(u)$ is a subset of $\texttt{BL}_{in}(x)$ such that $\texttt{BL}_{in}(x)$ will not be altered after we copy $\texttt{BL}_{in}(u)$ into $\texttt{BL}_{in}(x)$. $\texttt{BL}_{out}$ label will be updated in a similar manner but traversing from u in its ancestor direction.

It is noted that DL label covers only partial reachability information, it is thus possible that vertex v is reachable by vertex u if the condition in line 1 returns true. It appears the update becomes inefficient as redundant BFS is required for the label update. Nevertheless, the pruning conditions in line 7 will not enqueue vertices which cannot reach either u or v determined by BL label, which avoids visiting unnecessary vertices as many as possible. We show an example of the hash BL label as well as the insertion algorithm as the following.

Example 6. *Figure 7 shows the example of* BL *label update after edge insertion. The leaf nodes in our running example are* $\{1, 2, 3, 10, 11\}$. *Given a hash function* $h(\cdot)$, *assume* $h(1) = h(10) = 0$,

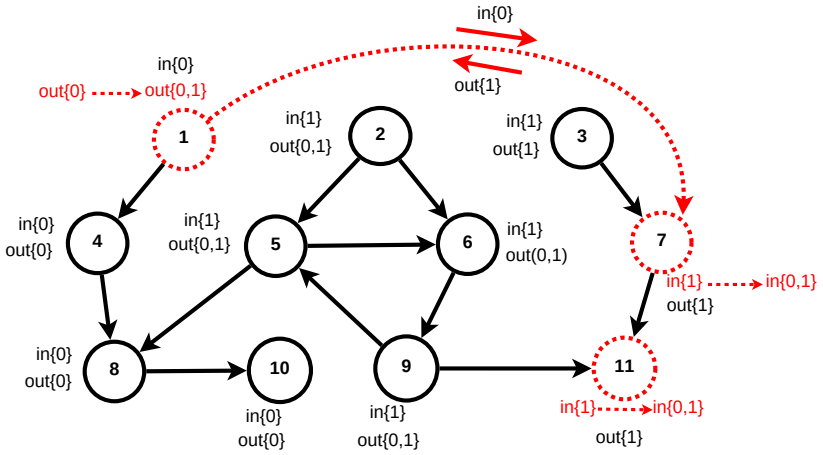


Figure 7: BL label update for inserting edge $(1, 7)$

$h(2) = h(3) = h(11) = 1$. *The hash* BL *label for each vertex can be easily translated using the hash function. If we insert edge* $(1, 7)$, $\texttt{BL}_{in}(1)$ *should be copied into all the* $\texttt{BL}_{in}(x)$ *where* x *belong to* $Des(7)$. *Thus* $\texttt{BL}_{in}(7) = \texttt{BL}_{in}(11) = \{0, 1\}$. *For the backward direction update,* $\texttt{BL}_{out}(7)$ *should be copied into all the* $\texttt{BL}_{out}(x)$ *where* x *belong to* $Anc(1)$. *Therefore,* $\texttt{BL}_{out}(1)$ *is updated to* $\{0, 1\}$.

When deleting edges from the graph, the update method for BL label is similar to that of DL label except the additional updates for the landmark vertices are not required. This is because, for DL label, the landmark node is a source to propagate its label and will always include itself as a label element in both $\texttt{DL}_{in}$ and $\texttt{DL}_{out}$ labels. For DL label update process, we need to make sure that the landmark itself will not be excluded from its DL label. For BL label, as all the label nodes are leaf nodes, the update process will not visit its label source node. Thus the inspection on source nodes' label is unnecessary. Hence, we only need to remove line 5-6, 16-17 in Algorithm 3 as the deletion algorithm for BL.

5.2 Discussions

We discuss some remaining issues for DL and BL label.

Vertex insertion/deletion. We treat a vertex update operation as a set of edge update operations. When inserting a new vertex, we create DL and BL labels followed by inserting all the related edges by the corresponding edge insertion algorithms. Likewise, for vertex deletion, we remove all its edges that are connected to the vertex and invoke the corresponding edge deletion algorithms.

Update complexity. In the worst case, all the vertices that reach or are reachable to the updating edges will be visited. Thus, the time complexity is $O((k + k')(m + n))$ where $(m + n)$ is the cost on BFS. Empirically, as the BFS procedure will prune a large number of vertices, the actual update process is much more efficient than a plain BFS.

Merge DL and BL update. Both BL label and DL label use BFS traversal to probe vertices for label updates. To reduce the number of random accesses incurred by BFS traversals, the update process of both labels are merged into a coalesced procedure. Take inserting (u, v) as an example, a BFS from v is used to update both $\texttt{DL}_{in}$ and $\texttt{BL}_{in}$ labels of vertices in $Des(v)$. Similarly, $\texttt{DL}_{out}$ and $\texttt{BL}_{out}$ labels of $Auc(u)$ are updated in another BFS from u. In this manner, we jointly update for BL and DL labels instead of launching unnecessary BFS that incurs additional random accesses to update the labels separately.

6 Experimental Evaluation

In this section, we conduct extensive experiments by comparing the proposed DBL framework with the state-of-the-art approaches on reachability query for dynamic graphs. We first introduce the

Table 2: Datasets statistics

Dataset	$\|V\|$	$\|E\|$	d_{avg}	Diameter	Connectivity(%)
LJ	4,847,571	68,993,773	14.23	16	78.9
Web	875,713	5,105,039	5.83	21	44.0
Email	265,214	420,045	1.58	14	13.8
Wiki	2,394,385	5,021,410	2.09	9	26.9
BerkStan	685,231	7,600,595	11.09	514	48.8
Pokec	1,632,803	30,622,564	18.75	11	80.0
Twitter	2,881,151	6,439,178	2.23	24	1.9
Reddit	2,628,904	57,493,332	21.86	15	69.2

Table 3: Datasets statistics for DAG

Dataset	DAG-$\|V\|$	DAG-$\|E\|$	DAG-INSERT (ms/edge)	DAG-DELETE (ms/edge)	DAG-CONSTRUCT (ms)
LJ	971,232	1,024,140	331	19449	2368
Web	371,764	517,805	132	1677	191
Email	231,000	223,004	21	66	17
Wiki	2,281,879	2,311,570	3524	167	360
BerkStan	109,406	583,771	6	1549	1134
Pokec	325,892	379,628	102	4386	86
Twitter	2,357,437	3,472,200	2173	64	481
Reddit	800,001	857,716	632	7498	1844

experimental setup and then report the results.

6.1 Experimental Setup

Compared Approaches: We compare the existing state-of-the-art solutions with the DBL framework. We obtain the implementations from their corresponding inventors.

- TOL [35]: A variant of the 2-hop label which optimizes the order of the label nodes. TOL relies on DAG to build its compact index with support of dynamic updates.
- IP [29]: The state-of-the-art solution for reachability query on dynamic graphs and it also relies on DAG. IP is based on min-wise independent permutation and answer the reachability by a set containment query.
- DBL: A sequential implementation of the proposed DBL framework.
- DBL-P: A OpenMP implementation of the proposed DBL framework on multi-cores. We follow the vertex-centric paradigm to parallelize DBL.
- DBL-G: A CUDA implementation of the proposed DBL framework on GPUs.

Datasets: We conduct experiments on 8 real-world datasets (see Table 2). LJ and Pokec are two social networks, which are power-law graphs in nature. BerkStan and Web are web graphs in which nodes represent web pages and directed edges represent hyperlinks between them. Wiki and Email are communication networks representing Wikipedia and Email respectively. The aforementioned datasets are collected from SNAP [16]. Reddit is an online forum and we crawl the interaction data between Reddit users for one month period. Twitter is an online social network and we crawl the tweets for one week via Twitter stream API. For each dataset, we record the average DAG maintenance time for a single edge insertion/deletion using the algorithms proposed in DAGGER [34]. The maintenance cost for each dataset as well as other network statistics can be found in Table 3. Apparently, DAG maintenance is a significant overhead and can be even slower than constructing DAG from scratch, especially for delete operation.

Table 4: Index construction time(s) and index size(MB)

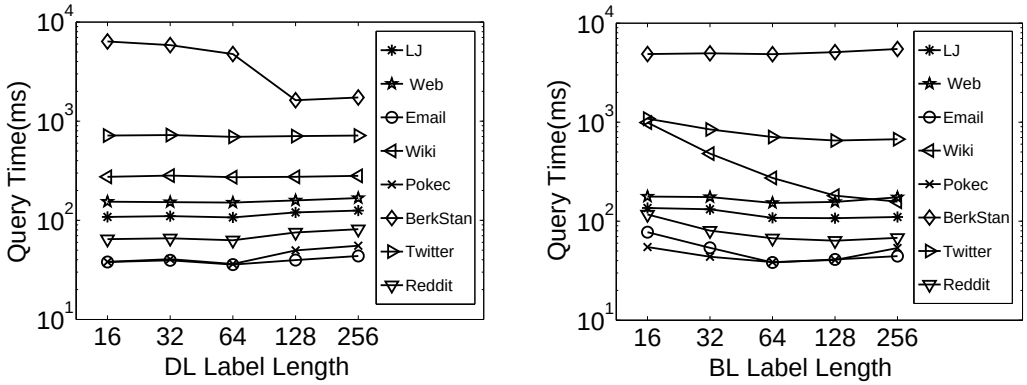
Dataset	Construction Time(s)			Index Size(MB)		
	TOL	IP	DBL	TOL	IP	DBL
LJ	4.678	2.79	114.92	3.81	15.21	147.93
Web	2.52	0.53	14.02	1.94	5.33	26.72
Email	0.29	0.09	0.91	0.83	2.57	8.09
Wiki	4.75	1.17	33.66	8.79	43.21	182.68
Pokec	0.81	0.22	49.89	3.01	5.23	49.83
BerkStan	3.09	1.21	3.36	6.93	1.45	31.37
Twitter	110.26	2.31	11.53	10.48	36.91	87.93
Reddit	3.06	2.15	88.67	3.11	13.28	80.23

Query on Dynamic Graph Setup: Since TOL and IP assume the SCCs in the graph remain unchanged after graph updates, the implementations we received from their inventors do not include DAG maintenance. To enable a reasonable comparison, we follow their experimental setups as the following: we randomly select 10000 edges from DAG of each dataset and perform DELETE as well as INSERT tests. For DELETE tests, we delete the selected edges and process 1 million reachability queries. For INSERT tests, we insert back the deleted edges and process 1 million queries. Under such test scenarios, TOL and IP could work as the underlying SCCs are not affected. We will report the index update time and the query processing time for both tests.

Environment: We implement our approaches in C++, and we adopt the C++ implementations of all baselines provided by their inventors. Our experiments are conducted on a server with an Intel Xeon CPU E5-2640 v4 2.4GHz, 256GB RAM and a Tesla P100 PCIe version GPU.

6.2 Index Construction

Table 4 reports the index construction time and index size for all compared approaches. TOL and IP are more efficient than DBL in terms of constructing the index since they build their index on DAG only. For index size, TOL requires the minimum space once it is a Label-Only approach. In contrast, IP and DBL are Label+G approaches thus require to store the graph as well as the labels for query processing. IP has a smaller size since it only stores DAG rather than the original graph for DBL. Nevertheless, we argue that this work focus on the dynamic scenario where index update is the key actor to consider. Thus, the index construction can be considered as an offline process. Moreover, the space of DBL label is linear to the graph size, and thus DBL scales for handling larger graphs.



(a) Varying DL label

(b) Varying BL label

Figure 8: Performance with varying label sizes

Table 5: Efficiency (ms) and percentages (%) of queries answered by DL and BFS when only DL is enabled

Dataset	DL Label	BFS	Total
LJ	36.71(97.5)	883.11(2.5)	921.55
Web	34.09(79.5)	1314.52(20.5)	1350.36
Email	14.99(31.9)	1568.26(68.1)	1585.02
Wiki	44.41(10.6)	14356.16(89.4)	14402.32
Pokec	40.03(97.6)	223.82(2.4)	267.49
BerkStan	27.09(87.5)	3256.01(12.5)	3306.22
Twitter	43.10(6.6)	20828.83(93.4)	20873.72
Reddit	30.98(93.7)	1233.82(6.3)	1266.53

Table 6: Efficiency (ms) and percentages (%) of queries answered by BL and BFS when only BL is enabled

Dataset	BL Label	BFS	Total
LJ	56.83(20.8)	–	–
Web	224.26(54.3)	–	–
Email	23.95(85.4)	187123.78(14.6)	187149.48
Wiki	130.90(94.3)	2619442.32(5.7)	2619585.79
Pokec	34.35(19.9)	–	–
BerkStan	223.69(43.3)	–	–
Twitter	36.25(94.8)	1175569.27(5.2)	1175607.29
Reddit	71.76(30.6)	–	–

6.3 Parameters in DBL

We study how the parameters affect the query processing of DBL (the parallel implementations of DBL are omitted since the data structures used for the index are essentially the same). There are two labels in DBL: both DL and BL store labels in bit vectors. The size of DL label depends on the number of selected landmark nodes whereas the size of BL label is determined by how many hash values are chosen to index the leaf nodes. We evaluate all the datasets to show the performance trend of varying DL and BL label sizes (process 1M queries) in Figure 8.

When varying DL label sizes k, the performance of most datasets remain stable before a certain size (e.g., 64 bits) and deteriorates thereafter. This means that extra landmark nodes will cover little extra reachability information. Thus, selecting more landmark nodes does not necessarily lead to better overall performance since the cost of processing the additional bits in the label increases. BerkStan will benefit from increasing the DL label size to 128 bits since 64 landmarks are not enough to cover enough reachable pairs.

Compared with DL label, most of the datasets will get a sweet spot when varying the size of BL label. This is because there are two conflicting factors which affect the overall performance. With more hash values incorporated, we can quickly prune more unreachable vertex pairs by examining BL label without traversing the graph. Besides, larger BL size also provides better pruning power of BFS even if it fails to directly answer the query (Algorithm 1). Nevertheless, the cost of label processing increases with more hash values in BL label. The only exception is Wiki. It shows a continuously speed up when BL label size increases. The benefit of larger BL label size outweighs the overheads of label processing.

In the remaining part of the experiments, we set Wiki's DL and BL label size as 64 and 256, BerkStan's DL and BL label size as 128 and 64. For the remaining datasets, both DL and BL label sizes are set as 64.

Table 7: Efficiency (ms) and percentages (%) of queries answered by DBL and BFS when both labels are enabled

Dataset	DBL	BFS	Total
LJ	30.99(99.8)	73.19(0.2)	105.91
Web	32.17(98.3)	105.41(1.7)	139.32
Email	16.09(99.2)	17.36(0.8)	35.17
Wiki	94.44(99.6)	61.24(0.4)	157.40
Pokec	24.96(99.9)	7.97(0.1)	34.67
BerkStan	60.03(95.0)	1673.40(5.0)	1591.21
Twitter	52.37(96.7)	667.44(3.3)	711.57
Reddit	33.08(99.9)	29.38(0.1)	64.21

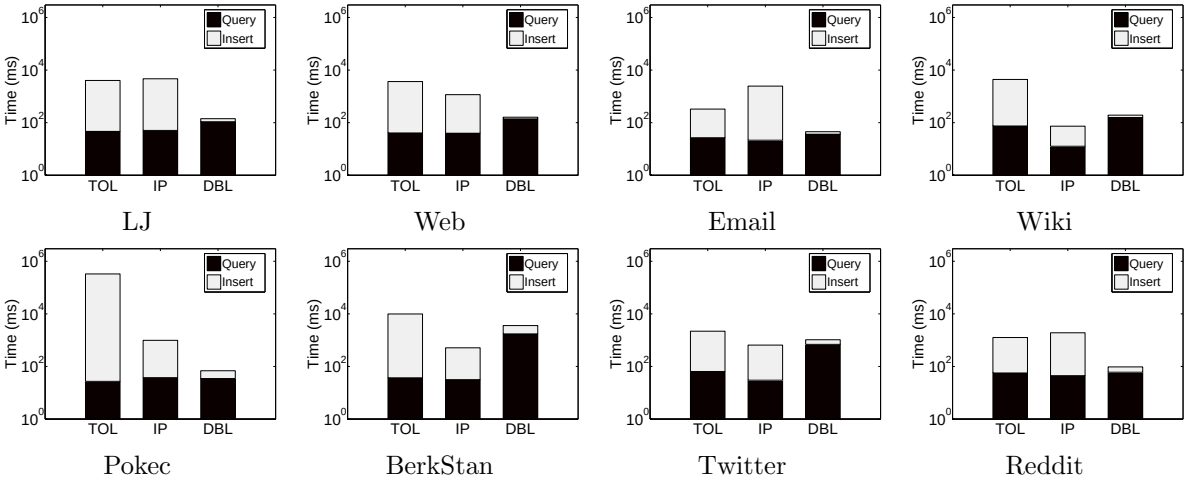


Figure 9: The execution time breakdown for INSERT tests.

6.4 Effectiveness of DL+BL

Table 5 and Table 6 show the query profiling results when only DL or BL label is enabled. The dash line in the table indicates that the query time is longer than an hour. In terms of percentages of queries handled by the labels only, the results show that DL is effective for dense graphs (LJ, Pokec and Reddit) whereas BL is effective for sparse graphs (Email, Wiki and Twitter). However, we still incur expensive BFS if either DL or BL is disabled. By combining the merits of both indexes, our proposal leads to a significantly better performance (Table 7). The results have validated our claim that DL and BL are complementary to each other.

6.5 Query on Dynamic Graph

In this section, we compare DBL with TOL and IP in terms of processing queries in dynamic graphs. For a fair comparison, we report the results for the sequential versions of the compared approaches and leave the discussions on parallel implementations in Section 6.6.

We show the overall performance of all solutions for INSERT and DELETE tests in Figure 9 and Figure 10 respectively. We want to highlight that the experiments in this subsection will not trigger DAG maintenance since TOL and IP cannot support DAG updates. The portion of time taken by query processing as well as index update for each compared approach are highlighted with different patterns.

For both tests, DBL outperforms IP and TOL in most cases. Although the baselines could provide better query processing performance than that of DBL, they incur expensive update costs and thus demonstrate inferior overall efficiency. On the contrary, DBL takes much shorter label update time due to the lightweight indexing approach as well as our efficient label update algorithms. For INSERT

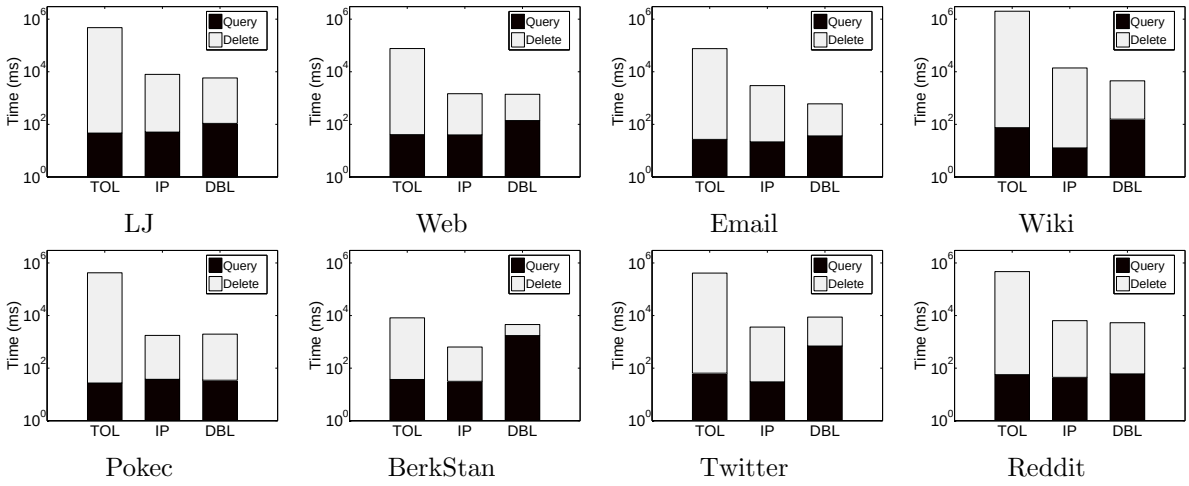


Figure 10: The execution time breakdown for `DELETE` tests.

updates, `DBL` provides up to 270x and 9700x speedups compared with `IP` and `TOL` respectively. For `DELETE` updates, `DBL` provides up to 5x and 533x speedups compared with `IP` and `TOL` respectively. It is also noted that `DELETE` update is a more complicated process than `INSERT` update for reachability indexes. Even though `IP` and `TOL` bring better query processing performance than `DBL`, their slow update efficiency results in a bottleneck and show unsatisfactory overall performance. Meanwhile, `DBL` offers competitive query processing performance. For one million reachability queries, `DBL` can answer them in sub-second performance for most datasets.

For BerkStan and Twitter, `DBL` is not the best solution. Moreover, the query processing as well as the index update on both datasets show inferior performance compared with the rest of the datasets. We have identified two critical properties from the graph structure (see Table 2) that affect the performance of query and index update: (1) the connectivity (probability of two randomly selected vertices are reachable); (2) the diameter. For query processing, `DL` label will be ineffective for poorly connected graphs since it is mainly used to determine reachable pairs. For graphs with low connectivity, we switch to `BL` label for pruning unreachable queries. If `BL` label is not effective as well, the query performance of `DBL` drops significantly as we have to invoke the pruned `BFS`. We have observed that a large diameter deteriorates the effectiveness of `BL` label. When constructing the index, `BFS` procedure is executed to propagate `BL` label. For graphs with a larger diameter, the propagation of `BL` label leads to higher collision rates for `BL` hash sets. When that happens, `BL` label is not effective for pruning unreachable queries and thus the overall performance is degraded. Take Bersktan for an example. It has a diameter of 514, which is much larger than the rest of the datasets. According to Table 7, Berkstan has the highest percentages of queries that are handled by pruned `BFS` across all datasets (5%). Besides, since its diameter is large, the pruned `BFS` also takes extra hops to determine the reachability. For Twitter, the dataset has the second largest diameter behind BerkStan and the graph is poorly connected (94% queries determined by `BL` label to be unreachable). This explains why the performance of Twitter is worse than that of Wiki, even though they share similar statistics except diameter (24 vs. 9) and connectivity (1.9% vs. 26.9%). For index update, diameter plays a key factor as well. The propagation of labels could penetrate the graph structure to a deeper depth for graphs with large diameters, e.g., BerkStan and Twitter, leading to inefficient update performance. Nevertheless, as shown in our additional experiments on real-world dynamic graphs in the appendix, our update mechanism achieves orders of magnitude speedups against `DAG` maintenance and processes thousands of updates per second.

We would like to also highlight two important issues here. First, both `TOL` and `IP` operate on `DAG`. As shown in Table 3, `DAG` has a much smaller size than the original graph. Since `DBL` directly works on the original graph without maintaining `DAG`, which, to some extent, explains why the two baselines could outperform `DBL` in terms of query processing since the underlying graphs are literally different. Second, we exclude `DAG` maintenance cost for `TOL` and `IP` since they do not support common dynamic

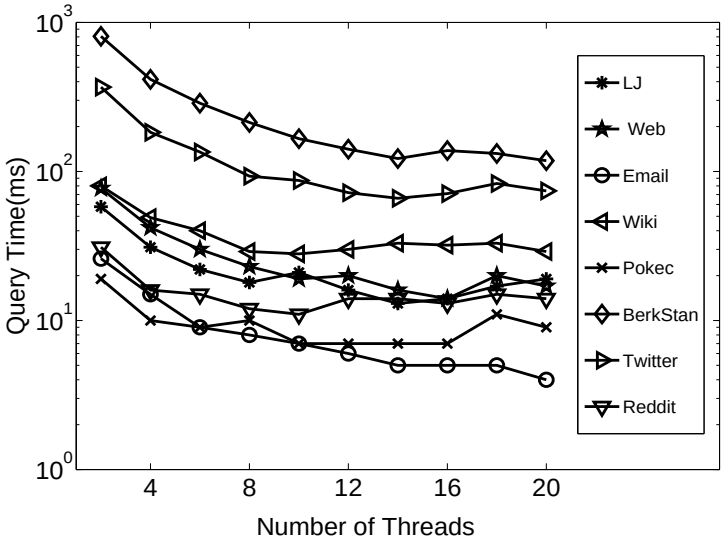


Figure 11: Scalability of DBL on CPU

graphs. However, this maintenance cost is substantial and overwhelms the rest of the processing tasks. We have studied additional real-world dynamic graph scenarios with DAG updates. We find that our approach is able to achieve 630-5700 times speedup against the state-of-the-art approach [34] for DAG updates. The detailed results can be found in Appendix 8.5.

6.6 Scalability and Parallelization

We implement DBL with OpenMP and CUDA (DBL-P and DBL-G respectively) to demonstrate the deployment on multi-core CPUs and GPUs achieves encouraging speedup. To validate the scalability of the parallel approach, we vary the number of threads used in DBL-P and show its performance trend in Figure 11. DBL-P achieves almost linear scalability against increasing number of threads (note that the y-axis is plotted in log-scale). The linear trend of scalability tends to disappear when the number of threads is beyond 14. We attribute this observation as the memory bandwidth bound nature of the processing tasks. DBL invokes BFS traversal once the labels are unable to answer the query and the efficiency of BFS is largely bounded by CPU memory bandwidth. This memory bandwidth bound issue of CPUs can be resolved by using GPUs which provide memory bandwidth boost.

We also compare our parallel solutions with an OpenMP implementation of IP (denoted as IP-P). The compared query processing performance is shown in Table 8. We note that IP has to invoke a pruned DFS if its labels fail to determine the query result. DFS is a sequential process in nature and

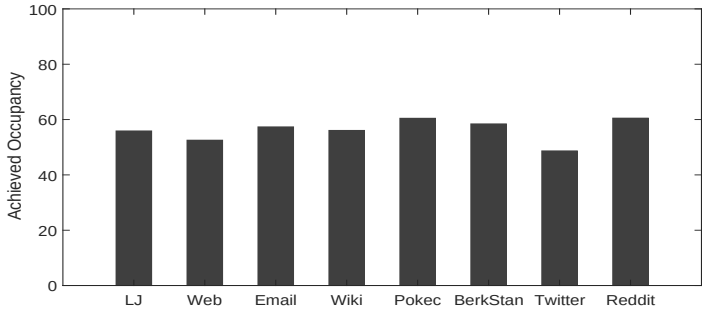


Figure 12: Warp Occupancy

Table 8: The query performance on CPU and GPU parallel architectures (ms)

Dataset	TOL	IP	IP-P	DBL	DBL-P	DBL-G	B-BFS
LJ	46.55	50.72	24.91	108	16.42	**6.03**	555561
Web	40.63	39.74	22.59	139	**12.36**	14.18	236892
Email	26.81	21.57	9.38	36.38	4.07	**2.77**	10168
Wiki	74.89	12.71	**4.10**	157	28.39	14.76	61113
Pokec	27.19	37.64	23.16	34.78	8.96	**3.05**	253936
BerkStan	37.16	31.64	**16.35**	1590	131	835	598127
Twitter	64.61	30.16	**7.14**	709	79.42	202	78496
Reddit	56.66	44.72	19.56	61.21	14.56	**3.07**	273935

cannot be efficiently parallelized. For our parallel implementation, we assign a thread to handle one query. As DFS incurs frequent random accesses, the performance is bounded by memory bandwidth. Thus, parallelization does not bring much performance gain compared with its sequential counterpart. In contrast, DBL is built on a pruned BFS which can be efficiently parallelized with vertex-centric paradigm. DBL provides competitive efficiency against IP-P while DBL-P demonstrates significant speedup over DBL. For example, DBL-P gets a 4x to 10x speedup across all datasets. DBL-G shows an even better performance. We note there are cases where DBL-P outperforms DBL-G, i.e., Web, Berkstan and Twitter. This is because these datasets have a higher diameter than the rest of the datasets and the pruned BFS needs to traverse extra hops to determine the reachability. Thus, we incur more random accesses for which is not favored by GPU architecture. We demonstrate the GPU utilization (average warp occupancy) of DBL-G in Figure 12. The utilization is consistently above 50 percent level and it shows that DBL can be efficiently accelerated on the GPU architecture.

7 Conclusion

In this work, we propose DBL, an indexing framework to support dynamic reachability query processing. DBL is the first solution which avoids maintaining DAG structure to construct and build reachability index. DBL leverages two complementary index components: DL and BL labels. DL label are built on the landmark nodes to determine reachable vertex pairs that connected by the landmarks, whereas BL label prunes unreachable pairs by examining their reachability information on the leaf nodes in the graph. In addition, the query processing and the index update procedure can be unified under BFS traversal and efficient parallel processing are readily available. The experimental evaluation has demonstrated that the sequential version of DBL outperforms the state-of-the-art solutions with orders of magnitude speedups in terms of index update while exhibits competitive query processing performance. The parallel implementation of DBL on multi-cores and GPUs further boost the performance over our sequential implementation.

8 Appendix

8.1 DL Label Construction

We show the batch construction of DL label in Algorithm 5. L is the landmark set used in DL construction. We first compute $M(\cdot)$ (lines 1-2) as the priority score of the vertex to be the landmark. Line 3 sorts all the vertices with a descend order of $M(\cdot)$ and the top k vertices are selected as the landmark set. The DL label construction process is bidirectional. Lines 6-13 construct $\texttt{DL}_{in}$ label and Line 14-21 construct $\texttt{DL}_{out}$ label. Take $\texttt{DL}_{in}$ label as an example, the BFS procedure will check whether the visited vertex is reachable by the label which has been built (line 11). If the vertex is determined to be reachable, it will not be enqueued as all the vertices in $Des(\cdot)$ will be reachable according to the DL label built. In other words, the corresponding reachability information of vertices in $Des(\cdot)$ have been covered by the previous processed landmarks and there is no need to cover the reachability information again. $\texttt{DL}_{out}$ label is constructed in a similar manner but we traversal from v_i on the reverse graph G'.

Algorithm 5 Dynamic Landmark Index Construction

Input: Graph $G(V, E)$, Landmark size k
Output: DL label for G

```
1: for all v ∈ V do
2:     Compute M(v) = |Pre(v)| · |Suc(v)|
3: L ← top-k nodes according to descending order of M
4: Q ← an empty queue
5: for i = 0; i < k; i++ do
6:     //Do BFS from L[i] in forward direction
7:     enqueue L[i] to Q
8:     while Q not empty do
9:         p ← pop Q
10:        for x ∈ Suc(p) do
11:            if DL_out(L[i]) ∩ DL_in(x) = ∅ then
12:                DL_in(x) ← DL_in(x) ∪ {L[i]};
13:                enqueue x to Q
14:    //Do BFS from L[i] in backward direction
15:    enqueue L[i] to Q
16:    while Q not empty do
17:        q ← pop Q
18:        for x ∈ Pre(q) do
19:            if DL_in(L[i]) ∩ DL_out(x) = ∅ then
20:                DL_out(x) ← DL_out(x) ∪ {L[i]};
21:                enqueue x to Q
```

Table 9: Query time for vertex pairs with different distances(ms)

Dataset	2-hop	4-hop	6-hop	8-hop	unreachable
LJ	6.99(99.98)	4.80(100)	3.10(100)	12.17(99.75)	59.97(99.02)
Web	13.42(98.36)	5.70(99.48)	3.87(100)	3.66(100)	30.45(97.03)
Email	4.42(98.83)	4.10(98.83)	2.67(99.73)	3.17(99.29)	4.77(99.09)
Wiki	13.84(99.85)	4.11(100)	28.33(99.05)	–	45.92(99.60)
Pokec	2.70(100)	4.49(100)	3.40(100)	3.69(99.79)	17.90(99.72)
BerkStan	11.1(97.81)	1.99(99.98)	5.04(99.30)	17.82(97.22)	322.05(90.12)
Twitter	9.36(99.99)	9.18(99.86)	34.26(98.91)	21.19(99.61)	87.46(96.64)
Reddit	4.00(100)	10.54(99.93)	32.25(98.77)	24.72(98.88)	25.68(99.56)

8.2 BL Label Construction

Algorithm 6 shows the BL label batch construction process. The process includes two BFS directions in a similar fashion. We will show the forward direction operation as an example. Firstly, we hash all the nodes with zero in-degrees to a bit vector of length k' (line 2-3).For each hash value i, we will collect all the nodes with the same hash value into the BFS queue respectively and jointly perform BFS to update the labels. For the nodes visited by the i_{th} BFS, the i_{th} bit of the $\texttt{BL}_{in}$ will be set 1 (line 10-11). The backward construction is almost the same except that it works on the reversed graph $G'(V, E')$ and constructs $\texttt{BL}_{out}$.e i_{th} bit of the $\texttt{BL}_{in}$ will be set 1 (line 7-11). The backward construction is almost the same except that it works on the reversed graph $G'(V, E')$ and constructs $\texttt{BL}_{out}$ (line 13-21).

8.3 BL Label Node Selection

In the main body of this paper, we restrict the leaf nodes to be the ones with either zero in-degree or zero out-degree. Nevertheless, our proposed method does not require such a restriction and could potentially select any vertex with low centrality as a leaf node. Following the approach for which

Algorithm 6 Bidirectional Leaf Label Construction

Input: Graph $G(V, E)$, Label size k'
Output: `BL` label for G

```
Initial all the bits in BL label as 0
for all v ∈ V with zero in-degrees do
    Hash v to [0, k' − 1]
Q ← an empty queue
for i from 0 to k' − 1 do
    for all nodes that hash value is i do
        Enqueue the nodes into Q for BFS
    //Construct the BL_in label
    Do BFS in forward direction
    for all nodes that BFS visited do
        Set the i_th bit of its BL_in 1.
    //Construct the BL_out label
    Do BFS in backward direction
    for all nodes that BFS visited do
        Set the i_th bit of its BL_out 1.
```

Table 10: Query time (ms) for different centrality heuristics. A=max($|Pre(\cdot)|, |Suc(\cdot|)$); B=min($|Pre(\cdot)|, |Suc(\cdot|)$); C=$|Pre(\cdot)| + |Suc(\cdot)|$; ours=$|Pre(\cdot)| \cdot |Suc(\cdot)|$

Dataset	A	B	C	ours
LJ	125.10	127.84	105.88	108.51
Web	202.16	144.13	142.16	139.64
Email	37.02	37.01	36.14	36.38
Wiki	156	159	153	157
Pokec	37.69	64.57	36.96	34.78
BerkStan	1890	6002	1883	1590
Twitter	719.31	849.78	685.31	693.71
Reddit	99.21	65.06	62.68	60.48

we select `DL` label nodes, we use $M(u) = |Pre(u)| \cdot |Suc(u)|$ to approximate the centrality of vertex u and select vertex u as a `BL` label node if $M(u) \leq r$ where r is a tunning parameter. Assigning $r = 0$ produces the special case presented in the main body of this paper. The algorithms for query processing as well as index update of the new `BL` label remains unchanged. Figure 13 shows the query performance of `DBL` when we vary the threshold r. With a higher r, more vertices are selected as the leaf nodes, which should theoretically improve the query processing efficiency. However, since we employ the hash function for `BL` label, more leaf nodes lead to higher collision rates. This explains why we don't observe a significant improvement in query performance.

8.4 Profiling of the Centrality Heuristic

Both `DL` and `BL` select the label nodes by heuristically approximating the centrality of a vertex u as $M(u) = |Pre(u)| \cdot |Suc(u)|$. In the appendix, we evaluate different heuristic methods for centrality approximation. The results are shown in Table 10. Overall, our adopted heuristic ($|Pre(u)| \cdot |Suc(u)|$) achieves the performance. For Email and Wiki, all the methods share a similar performance. $|Pre(\cdot)| + |Suc(\cdot)|$ and $|Pre(\cdot)| \cdot |Suc(\cdot)|$ get a better performance in other datasets. Finally, $|Pre(\cdot)| + |Suc(\cdot)|$ and $|Pre(\cdot)| \cdot |Suc(\cdot)|$ deliver similar performance for most datasets and the latter is superior in the Berkstan dataset. Thus, we adopt $|Pre(\cdot)| \cdot |Suc(\cdot)|$ for approximating the centrality.

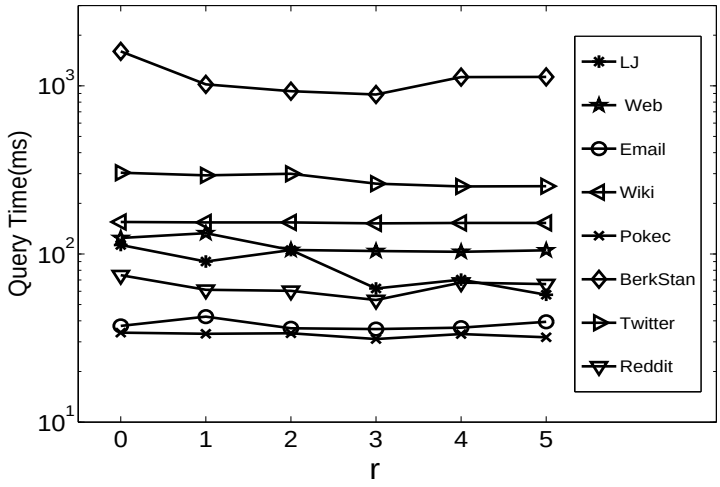


Figure 13: BL label node selection

8.5 Update for Real World Graph

We report additional results on real-world dynamic graphs: sx-stackoverflow (SX) and wiki-talk-temporal (WTALK) from SNAP [16]. WTALK consists of 1,140,149 vertices and 7,833,140 edges and SX is a much larger graph which has 2,601,977 vertices and 63,497,050 edges. As each edge has a timestamp, we order the edges according to their timestamps and run a sliding window which contains the first 50% of edges in each dataset to maintain a temporal graph that gets updated once the window slides. Table 11 presents the results for SX and WTALK. We report the number of DAG updates and the running time of processing edge updates after every 10000 window slides. As both IP and TOL can't handle real-world update, we use DAGGER as baseline instead. The results have shown that the number of DAG updates is significant and cannot be ignored. For example, there are 2471 and 3844 DAG updates for 50000 window slides. However, it takes hours to process the updates. In comparison, DBL completes the index update in 48.71 and 2.81 seconds.

Table 11: Profiling of the real-world graph updates

	SX			WTALK		
Slides	DAGGER (s)	DBL (s)	DAG updates	DAGGER (s)	DBL (s)	DAG updates
10000	13417	21.65	469	1912	0.93	691
20000	27434	28.42	939	4021	1.39	1378
30000	46420	35.17	1365	6801	1.86	2013
40000	66246	40.94	1838	10967	2.35	2810
50000	89846	48.71	2471	14036	2.81	3844
60000	110312	55.51	3054	16907	3.24	4640
70000	131628	62.28	3534	20005	3.66	5571
80000	154159	68.13	4012	23151	4.08	6451

8.6 Vary Vertex Pair Distance

To further validate the effectiveness of DBL, we evaluate the query performance of the node pairs in different distances. For each scenario, One hundred thousand queries are generated in a totally random manner. Table 9 shows the query time. The number in bracket is the percentage of the query directly answered by the labels. The dash line means that we can't generate enough queries. In fact, for pairs of vertices that are 2-hops away, 4-hops away or 6-hops away, the query performance will make

Table 12: Insert time for vertex pairs with different distances(ms)

Dataset	2-hop	4-hop	6-hop	8-hop	unreachable
LJ	15.91(98.64)	15.31(97.47)	12.07(85.20)	25.94(61.77)	62.89(0.23)
Web	12.38(66.42)	17.60(52.30)	9.51(78.14)	8.87(79.03)	31.56(1.10)
Email	6.53(2.27)	7.90(14.91)	7.69(0.77)	8.52(1.28)	10.36(0.26)
Wiki	94.62(28.21)	9.72(98.72)	15.42(1.92)	3.03(1.39)	399.72(0)
Pokec	12.67(99.21)	16.61(98.74)	13.22(57.87)	5.38(85.80)	59.96(0.47)
BerkStan	5.93(90.38)	16.72(89.72)	9.31(0)	5.46(0)	2437.07(22.50)
Twitter	55.11(92.45)	162.48(74.29)	32.11(34.36)	28.01(1.43)	4371.52(0)
Reddit	22.57(98.70)	119.21(0)	28.31(0)	9.91(38.16)	170.15(0.22)

a difference only when the `BFS` procedures are required. Generally, a longer distance tends to cost a larger overhead as the `BFS` procedure will need more time to traversal the graph. However, the percentage of queries that answered by label will also have a great impact on the total query time which could explain the fluctuation in the query performance.

We also evaluate the edge insert performance for node pairs with different distances. The result is shown in Table 12. In order to be consistent with the preceding setting, ten thousands insertion are executed. According to Algorithm 2 and Algorithm 4, the `DL` pruning efficiency(line 1) will have a great impact on the insert performance. We list the percentage of the index update pruned by the `DL` label in the bracket. As `DL` label only cover part of the reachability information, some of the reachable vertex pair may be enqueued, however, the `BFS` procedure will be early terminated as the index is unaffected by the edge insertion.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. *Efficient management of transitive relationships in large data and knowledge bases*, volume 18. ACM, 1989.

[2] 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.

[3] R. Bramandia, B. Choi, and W. K. Ng. Incremental maintenance of 2-hop labeling of large graphs. *TKDE*, 22(5):682–698, 2010.

[4] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *CIKM*, pages 119–128. ACM, 2010.

[5] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, pages 893–902. IEEE, 2008.

[6] 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 *SIGMOD*, pages 193–204. ACM, 2013.

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

[8] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *FOCS*, pages 260–267. IEEE, 2001.

[9] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *CIKM*, pages 594–601. ACM, 2005.

[10] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *FOCS*, pages 664–672. IEEE, 1995.

[11] H. Jagadish. A compression technique to materialize transitive closure. *TODS*, 15(4):558–598, 1990.

[12] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *TODS*, 36(1):7, 2011.

[13] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14):1978–1989, 2013.

[14] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826. ACM, 2009.

[15] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608. ACM, 2008.

[16] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[18] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 867–876. ACM, 2009.

[19] L. Roditty. Decremental maintenance of strongly connected components. In *SODA*, pages 1143–1150. SIAM, 2013.

[20] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SICOMP*, 45(3):712–733, 2016.

[21] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371. IEEE, 2005.

[22] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pages 1009–1020. IEEE, 2013.

[23] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *SIGPLAN*, volume 48, pages 135–146. ACM, 2013.

[24] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: can it be even faster? *TKDE*, (1):1–1, 2017.

[25] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856. ACM, 2007.

[26] R. R. Veloso, L. Cerf, W. Meira Jr, and M. J. Zaki. Reachability queries in very large graphs: A fast refined online search approach. In *EDBT*, pages 511–522. Citeseer, 2014.

[27] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, pages 75–75. IEEE, 2006.

[28] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *PVLDB*, 7(12):1191–1202, 2014.

[29] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: an independent permutation labeling approach. *VLDBJ*, 27(1):1–26, 2018.

[30] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *GRADES*, page 2. ACM, 2013.

[31] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606. ACM, 2013.

[32] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1-2):276–284, 2010.

[33] H. Yıldırım, V. Chaoji, and M. J. Zaki. Grail: a scalable index for reachability queries in very large graphs. *VLDBJ*, 21(4):509–534, 2012.

[34] H. Yildirim, V. Chaoji, and M. J. Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *Computer Science*, 21(4):509–534, 2013.

[35] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*, pages 1323–1334. ACM, 2014.