4-2021

DBL: Efficient reachability queries on dynamic graphs

Qiuyi LYU
Shandong University

Yuchen LI
Singapore Management University, yuchenli@smu.edu.sg

Bingsheng HE

Bin GONG

DBL: Efficient Reachability Queries on Dynamic Graphs (Complete Version)

Qiuyi Lyu[1], Yuchen Li[2], Bingsheng He[3], and Bin Gong[4]

[1] Shandong University qiuyilv@gmail.com
[2] Singapore Management University yuchenli@smu.edu.sg
[3] National University of Singapore hebs@comp.nus.edu.sg
[4] Shandong University gb@sdu.edu.cn

Abstract. Reachability query is a fundamental problem on graphs, which has been extensively studied in academia and industry. Since graphs are subject to frequent updates in many applications, it is essential to support efficient graph updates while offering good performance in reachability queries. Existing solutions compress the original graph with the Directed Acyclic Graph (`DAG`) and propose efficient query processing and index update techniques. However, they focus on optimizing the scenarios where the Strong Connected Components (`SCC`s) remain unchanged and have overlooked the prohibitively high cost of the `DAG` maintenance when `SCC`s are updated. In this paper, we propose `DBL`, an efficient `DAG`-free index to support the reachability query on dynamic graphs with insertion-only updates. `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 latter prunes unreachable pairs by indexing the leaf nodes in the graph. 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 `DBL` achieves orders of magnitude speedup in terms of index update, while still producing competitive query efficiency.

1 Introduction

Given a graph G and a pair of vertices u and v, reachability query (denoted as $q(u, v)$) is a fundamental graph operation that answers whether there exists a path from u to v on G. This operation is a core component in supporting numerous applications in practice, such as those in social networks, biological complexes, knowledge graphs, and transportation networks. A plethora of index-based approaches have been developed over a decade [24,5,22,17,25,18,20,26,21,9] and demonstrated great success in handling reachability query on *static* graphs with millions of vertices and edges. However, in many cases, graphs are highly *dynamic* [23]: New friendships continuously 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 and temporary traffic controls occur. In those applications, it is essential to support efficient graph updates while offering good performance in reachability queries.

There have been some efforts in developing reachability index to support graph updates [4,6,8,10,15,16,17]. However, there is a major assumption made in those works: the Strongly Connected Components (SCCs) in the underlying graph remain unchanged after the graph gets updated. The Directed Acyclic Graph (DAG) collapses the SCCs into vertices and the reachability query is then processed on a significantly smaller graph than the original. The state-of-the-art solutions [27,24] thus rely on the DAG to design an index for efficient query processing, yet their index maintenance mechanisms only support the update which does not trigger SCC merge/split in the DAG. However, such an assumption can be invalid in practice, as edge insertions could lead to updates of the SCCs in the DAG. In other words, the overhead of the DAG maintenance has been mostly overlooked in the previous studies.

One potential solution is to adopt existing DAG maintenance algorithms such as [26]. Unfortunately, this DAG maintenance is a prohibitively time-consuming process, as also demonstrated in the experiments. For instance, in our experiments, the time taken to update the DAG on one edge insertion in the LiveJournal dataset is two-fold more than the time taken to process *1 million queries* for the state-of-the-art methods. Therefore, we need a new index scheme with a low maintenance cost while efficiently answering reachability queries.

In this paper, we propose a DAG-free dynamic reachability index framework(DBL) that enables efficient index update and supports fast query processing at the same time on large scale graphs. We focus on insert-only dynamic graphs with new edges and vertices continuously added. This is because the number of deletions are often significantly smaller than the number of insertions, and deletions are handled with lazy updates in many graph applications [2,3]. Instead of maintaining the DAG, we index the reachability information around two sets of vertices: the "landmark" nodes with high centrality and the "leaf" nodes with low centrality (e.g., nodes with zero in-degree or out-degree). As the reachability information of the landmark nodes and the leaf nodes remain relatively stable against graph updates, it enables efficient index update opportunities compared with approaches using the DAG. Hence, DBL is built on the top of two simple and effective index components: (1) a Dynamic Landmark (DL) label, and (2) a Bidirectional Leaf (BL) label. Combining DL and BL in the DBL ensures efficient index maintenance while achieves competitive query processing performance.

Efficient query processing: DL is inspired by the landmark index approach [5]. The proposed DL label maintains a small set of the landmark nodes as the label for each vertex in the graph. Given a query $q(u, v)$, if both the DL labels of u and v contain a common landmark node, we can immediately determine that u reaches v. Otherwise, we need to invoke Breadth-First Search(BFS) to process $q(u, v)$. We 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 form an exclusive set apart from the landmark node set. BL label of a vertex u is defined to be the leaf nodes which can either reach u or u can reach them. Hence, u does not reach v if there exists one leaf node in u's BL label which does not appear in the BL label of v. In summary, DL can quickly determine reachable pairs while BL, which complements DL, prunes disconnected pairs to remedy the ones that cannot be immediately determined by DL.

Efficient index maintenance: Both DL and BL labels are lightweight indexes where each vertex only stores a constant size label. When new edges are inserted, efficient pruned BFS is employed and only the vertices where their labels need update will be visited. In particular, once the label of a vertex is unaffected by the edge updates, we safely prune the vertex as well as its descendants from the BFS, which enables efficient index update.

To better utilize the computation power of modern architectures, we implement DL and BL with simple and compact bitwise operations. Our implementations are based on OpenMP and CUDA in order to exploit parallel architectures multi-core CPUs and GPUs (Graphics Processing Units), respectively.

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 the DAG. In addition, the algorithms can be easily implemented with parallel interfaces.
- We conduct extensive experiments to validate the performance of DBL in comparison with the state-of-the-art dynamic methods [27,24]. DBL achieves competitive query performance and orders of magnitude 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 preliminaries and background. Section 3 presents the related work. Section 4 presents the index definition as well as query processing. Sections 5 demonstrate the update mechanism of DL and BL labels. Section 6 reports the experimental results. Finally, we conclude the paper in Section 7.

2 Preliminaries

A directed graph is defined as $G = (V, E)$, where V is the vertex set and E is the edge set with $n = |V|$ and $m = |E|$. 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 that v is reachable by u when there exists a $Path(u, v)$ in G. 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

Table 1: Common notations in this paper

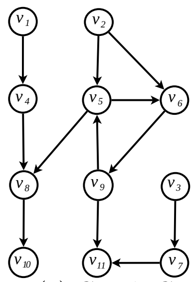
Notation	Description
$G(V, E)$	the vertex set V and the edge set E of a directed graph G
G'	the reverse graph of G
n	the number of vertex in G
m	the number of edges in G
$Suc(u)$	the set of u's out-neighbors
$Pre(u)$	the set of u's in-neighbors
$Des(u)$	the set of u's descendants including u
$Anc(u)$	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 size of DL label for one vertex
k'	the size of BL label for one vertex
$\mathtt{DL}_{in}(u)$	the label that keeps all the landmark nodes that could reach u
$\mathtt{DL}_{out}(u)$	the label that keeps all the landmark nodes that could be reached by u
$\mathtt{BL}_{in}(u)$	the label that keeps the hash value of the leaf nodes that could reach u
$\mathtt{BL}_{out}(u)$	the label that keeps the hash value of the leaf nodes that could be reached by u
$h(u)$	the hash function that hash node u to a value

u (including u) as $Anc(u)$ and all the descendants of u (including u) as $Des(u)$. We denote the reversed graph of G as $G' = (V, E')$ where all the edges of G are in the opposite direction of 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 $q(u, v)$ as a reachability query from u to v. In this paper, we study the dynamic scenario where edges can be inserted into the graph. Common notations are summarized in Table 1.

3 Related Work

There have been some studies on dynamic graph [4,6,8,10,15,16,17]. Yildirim et al. propose DAGGER [26] which maintains the graph as a DAG after insertions and deletions. The index is constructed on the DAG to facilitate reachability query processing. The main operation for the DAG maintenance is the merge and split of the *Strongly Connected Component* (SCC). Unfortunately, it has been shown that DAGGER exhibits unsatisfactory query processing performance on handling large graphs (even with just millions of vertices [27]).

The state-of-the-art approaches: TOL [27] and IP [24] follow the maintenance method for the DAG from DAGGER and propose novel dynamic index on the DAG to improve the query processing performance. We note that TOL and IP are only applicable to the scenarios where the SCC/s in the DAG remains unchanged against updates. In the case of SCC merges/collapses, DAGGER is still required to recover the SCC/s. For instance, TOL and IP can handle edge insertions (v_1, v_5) in Figure 1(a), without invoking DAGGER. However, when inserting (v_9, v_2), two SCC/s $\{v_2\}$ and $\{v_5, v_6, v_9\}$ will be merged into one larger SCC $\{v_2, v_5, v_6, v_9\}$. For such cases, TOL and IP rely on DAGGER for maintaining the DAG first and then perform their respective methods for index maintenance and query processing.



(a) Graph G

v	$\mathtt{DL}_{in}$	$\mathtt{DL}_{out}$
v_1	$\emptyset$	$\{v_8\}$
v_2	$\emptyset$	$\{v_5, v_8\}$
v_3	$\emptyset$	$\emptyset$
v_4	$\emptyset$	$\{v_8\}$
v_5	$\{v_5\}$	$\{v_5, v_8\}$
v_6	$\{v_5\}$	$\{v_5, v_8\}$
v_7	$\emptyset$	$\emptyset$
v_8	$\{v_5, v_8\}$	$\{v_8\}$
v_9	$\{v_5\}$	$\{v_5, v_8\}$
v_{10}	$\{v_5, v_8\}$	$\emptyset$
v_{11}	$\{v_5\}$	$\emptyset$

(b) `DL` label for G

v	$\mathtt{BL}_{in}$	$h(\mathtt{BL}_{in})$	$\mathtt{BL}_{out}$	$h(\mathtt{BL}_{out})$
v_1	$\{v_1\}$	$\{0\}$	$\{v_{10}\}$	$\{0\}$
v_2	$\{v_2\}$	$\{1\}$	$\{v_{10}, v_{11}\}$	$\{0,1\}$
v_3	$\{v_3\}$	$\{1\}$	$\{v_{11}\}$	$\{1\}$
v_4	$\{v_1\}$	$\{0\}$	$\{v_{10}\}$	$\{0\}$
v_5	$\{v_2\}$	$\{1\}$	$\{v_{10}, v_{11}\}$	$\{0,1\}$
v_6	$\{v_2\}$	$\{1\}$	$\{v_{10}, v_{11}\}$	$\{0,1\}$
v_7	$\{v_3\}$	$\{1\}$	$\{v_{11}\}$	$\{1\}$
v_8	$\{v_1, v_2\}$	$\{0,1\}$	$\{v_{10}\}$	$\{0\}$
v_9	$\{v_2\}$	$\{1\}$	$\{v_{10}, v_{11}\}$	$\{0,1\}$
v_{10}	$\{v_1, v_2\}$	$\{0,1\}$	$\{v_{10}\}$	$\{0\}$
v_{11}	$\{v_2, v_3\}$	$\{1\}$	$\{v_{11}\}$	$\{1\}$

(c) `BL` label for G

Fig. 1: A running example of graph G

However, the overheads of the `SCC` maintenance are excluded in their experiments [27,24] and such overheads is in fact non-negligible [26,14].

In this paper, we propose the `DBL` framework which only maintains the labels for all vertices in the graph without constructing the `DAG`. That means, `DBL` can effectively avoid the costly `DAG` maintenance upon graph updates. `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 updates.

4 DBL Framework

The `DBL` framework is consist of `DL` and `BL` label which have their independent query and update components. In this section, we introduce the `DL` and `BL` label. Then, we devise the query processing algorithm that builds upon `DBL` index.

4.1 Definitions and Construction

We propose the `DBL` framework that consists of two index components: `DL` and `BL`.

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

It is noted that `DL` label is a subset of the 2-hop label [5]. In fact, 2-Hop label is a special case for `DL` label when the landmark set $L = V$. Nevertheless, we find that maintaining 2-Hop label in the dynamic graph scenario leads to index explosion. Thus, we propose to only choose a subset of vertices as the landmark set L to index `DL` label. In this way, `DL` label has up to $O(n|L|)$ space complexity and the index size can be easily controlled by tunning the selection of L. The following lemma shows an important property of `DL` label for reachability query processing.

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

Example 1. We show an running example in Figure 1(a). Assuming the landmark set is chosen as $\{v_5, v_8\}$, the corresponding DL label is shown in Figure 1(b). $q(v_1, v_{10})$ returns true since $\mathtt{DL}_{out}(v_1) \cap \mathtt{DL}_{in}(v_{10}) = \{v_8\}$. However, the labels cannot give negative answer to $q(v_3, v_{11})$ despite $\mathtt{DL}_{out}(v_3) \cap \mathtt{DL}_{in}(v_{11}) = \emptyset$. This is because the intermediate vertex v_7 on the path from v_3 to v_{11} is not included in the landmark set.

To achieve good query processing 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., $\mathtt{DL}_{out}(u) \cap \mathtt{DL}_{in}(v)$ 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 [13]. In this paper, we adopt a heuristic method for selecting DL label nodes following existing works [1,13]. In particular, we rank vertices with $M(u) = |Pre(u)| \cdot |Suc(u)|$ to approximate their centrality and select top-k vertices. Other landmark selection methods are also discussed in Section 6.2.

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

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

BL label can give 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. DL label is efficient for giving positive answer to a reachability query whereas BL label plays a complementary role by pruning unreachable pairs. In this paper, we take vertices with zero in-degree/out-degree as the leaf nodes. We also discuss other leaf selection methods in Section 6.2.

Example 2. Figure 1(c) shows BL label for the running example. BL label gives negative answer to $q(v_4, v_6)$ since $\mathtt{BL}_{in}(v_4)$ is not contained by $\mathtt{BL}_{in}(v_6)$. Intuitively, vertex v_1 reaches vertex v_4 but cannot reach v_6 which indicates v_4 should not reach v_6. BL label cannot give positive answer. Take $q(v_5, v_2)$ for an example, the labels satisfy the containment condition but positive answer cannot be given.

The number of BL label nodes could be huge. To develop efficient index operations, we build a hash set of size k' for BL as follows. Both $\mathtt{BL}_{in}$ and $\mathtt{BL}_{out}$ are a subset of $\{1, 2, \dots, k'\}$ where k' is a user-defined label size, and they are stored in bit vectors. A hash function is used to map the leaf nodes to a corresponding bit. For our example, the leaves are $\{v_1, v_2, v_3, v_{10}, v_{11}\}$. When $k' = 2$, all leaves are hashed to two unique values. Assume $h(v_1) = h(v_{10}) = 0$, $h(v_2) = h(v_3) = h(v_{11}) = 1$. We show the hashed BL label set in Figure 1(c) which are denoted

Algorithm 1 DL label Batch Construction

Input: Graph $G(V, E)$, Landmark Set D
Output: DL label for G
1: **for** $i = 0$; $i < k$; $i{+}{+}$ **do**
2: //Forward BFS
3: $S \leftarrow D[i]$
4: enqueue S to an empty queue Q
5: **while** Q not empty **do**
6: $p \leftarrow$ pop Q
7: **for** $x \in Suc(p)$ **do**
8: $\mathtt{DL}_{in}(x) \leftarrow \mathtt{DL}_{in}(x) \cup \{S\}$;
9: enqueue x to Q
10: //Symmetrical Backward BFS is performed.

as $h(\mathtt{BL}_{in})$ and $h(\mathtt{BL}_{out})$. In the rest of the paper, we directly use $\mathtt{BL}_{in}$ and $\mathtt{BL}_{out}$ to denote the hash sets of the corresponding labels. It is noted that one can still use Lemma 2 to prune unreachable pairs with the hashed BL label.

We briefly discuss the batch index construction of DBL as the focus of this work is on the dynamic scenario. The construction of DL label is presented in Algorithm 1, which follows existing works on 2-hop label [5]. For each landmark node $D[i]$, we start a BFS from S (Line 4) and include S in $\mathtt{DL}_{in}$ label of every vertices that S can reach (Lines 5-9). For constructing $\mathtt{DL}_{out}$, we execute a BFS on the reversed graph G' symmetrically (Line 10). To construct BL label, we simply replace the landmark set D as the leaf set D' and replace S with all leaf nodes that are *hashed* to bucket i (Line 3) in Algorithm 1. The complexity of building DBL is that $O((k + k')(m + n))$.

Note that although we use [5] for offline index construction, the contribution of our work is that we construct DL and BL as complementary indices for efficient query processing. Furthermore, we are the first work to support efficient dynamic reachability index maintenance without assuming SCC/s remain unchanged.

Space complexity. The space complexities of DL and BL labels are $O(kn)$ and $O(k'n)$, respectively.

4.2 Query Processing

With the two indexes, Algorithm 2 illustrates the query processing framework of DBL. Given a reachability query $q(u, v)$, we return the answer immediately if the labels are sufficient to determine the reachability (Lines 6-9). By the definitions of DL and BL labels, u reaches v if their DL label overlaps (Line 6) where u does not reach v if their BL label does not overlap (Line 9). Furthermore, there are two early termination rules implemented in Lines 10 and 12, respectively. Line 10 makes use of the properties that all vertices in a SCC contain at least one common landmark node. Line 12 takes advantage of the scenario when either u or v share the same SCC with a landmark node l then u reaches v if and only if l appeared in the DL label of u and v. We prove their correctness in Theorem

Algorithm 2 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: return $(\mathtt{DL}_{out}(x) \cap \mathtt{DL}_{in}(y))$;
3: **function** BL_$Contain(x$,$y)$
4: return $(\mathtt{BL}_{in}(x) \subseteq \mathtt{BL}_{in}(y)$ and $\mathtt{BL}_{out}(y) \subseteq \mathtt{BL}_{out}(x))$;
5: **procedure** QUERY$(u$,$v)$
6: **if** DL_$Intersec(u$,$v)$ **then**
7: return true;
8: **if** not BL_$Contain(u$,$v)$ **then**
9: return false;
10: **if** DL_$Intersec(v$,$u)$ **then**
11: return false;
12: **if** DL_$Intersec(u$,$u)$ or DL_$Intersec(v$,$v)$ **then**
13: return false;
14: Enqueue u for BFS;
15: **while** queue not empty **do**
16: $w \leftarrow$ pop queue;
17: **for** vertex $x \in Suc(w)$ **do**
18: **if** $x = v$ **then**
19: return true;
20: **if** DL_$Intersec(u$,$x)$ **then**
21: continue;
22: **if** not BL_$Contain(x$,$v)$ **then**
23: continue;
24: Enqueue x;
25: **return** false;

1 and Theorem 2 respectively. Otherwise, we turn to BFS search with efficient pruning. The pruning within BFS is performed as follows. Upon visiting a vertex q, the procedure will determine whether the vertex q should be enqueued in Lines 20 and 22. BL and DL labels will judge whether the destination vertex v will be in the $Des(w)$. If not, q will be pruned from BFS to quickly answer the query before traversing the graph with BFS.

Theorem 1. *In Algorithm 2, when* DL_$Intersec(x$,$y)$ *returns false and* DL_$Intersec(y$,$x)$ *returns true, then x cannot reach y.*

Proof. DL_$Intersec(y, x)$ returns true indicates that vertex y reaches x. If vertex x reaches vertex y, then y and x must be in the same SCC (according to the definition of the SCC). As all the vertices in the SCC are reachable to each other, the landmark nodes in $\mathtt{DL}_{out}(y) \cap \mathtt{DL}_{in}(x)$ should also be included in $\mathtt{DL}_{out}$ and $\mathtt{DL}_{in}$ 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.

Algorithm 3 $\texttt{DL}_{in}$ label update for edge insertion

Input: Graph $G(V, E)$, $\texttt{DL}$ label, Inserted edge (u, v)
Output: Updated $\texttt{DL}$ label
1: **if** $\texttt{DL}_{out}(u) \cap \texttt{DL}_{in}(v) == \emptyset$ **then**
2: Initialize an empty queue and enqueue v
3: **while** queue is not empty **do**
4: $p \leftarrow$ pop queue
5: **for** vertex $x \in Suc(p)$ **do**
6: **if** $\texttt{DL}_{in}(u) \not\subseteq \texttt{DL}_{in}(x)$ **then**
7: $\texttt{DL}_{in}(x) \leftarrow \texttt{DL}_{in}(x) \cup \texttt{DL}_{in}(u)$
8: enqueue x

Theorem 2. *In Algorithm 2, if* $\texttt{DL}\_Intersec(x, y)$ *returns false and* $\texttt{DL}\_Intersec(x,x)$ *or* $\texttt{DL}\_Intersec(y,y)$ *returns true then vertex x cannot reach y.*

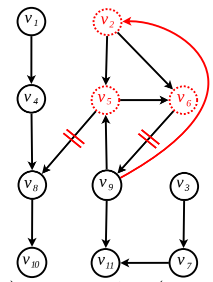
Proof. If $\texttt{DL}\_Intersec(x,x)$ returns true, it means that vertex x is a landmark or x is in the same SCC with a landmark. If x is in the same SCC with landmark l, vertex x and vertex l should have the same reachability information. As landmark l will push its label element l to $\texttt{DL}_{out}$ 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 $\texttt{DL}\_Intersec(x, y)$ returns false. The proving process is similar for the case when $\texttt{DL}\_Intersec(y, y)$ 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 the pruned BFS search, which has a worst case time complexity of $O((k+k')(m + n))$. Let ρ denote the ratio of vertex pairs whose reachability could be directly answered by the label. The amortized time complexity is $O(\rho(k + k') + (1 - \rho)(k + k')(m + n))$. Empirically, ρ is over 95% according to our experiments (Table 4 in Section 6), which implies efficient query processing.

5 DL and BL Update for Edge Insertions

When inserting a new edge (u, v), all vertices in $Anc(u)$ can 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 update the label by 1) adding $\texttt{DL}_{in}(u)$ into $\texttt{DL}_{in}(x)$ for all $x \in Des(v)$; and 2) adding $\texttt{DL}_{out}(v)$ into $\texttt{DL}_{out}(x)$ for all $x \in Anc(u)$.

Algorithm 3 depicts the edge insertion scenario for $\texttt{DL}_{in}$. We omit the update for $\texttt{DL}_{out}$, which is symmetrical to $\texttt{DL}_{in}$. 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 (Line 1). Lines 2-8 describe a BFS



(a) Insert edge (v_9, v_2)

v	$\mathtt{DL}_{in}$
v_1	$\emptyset \to \emptyset$
v_2	$\emptyset \to \{v_5\}$
v_3	$\emptyset \to \emptyset$
v_4	$\emptyset \to \emptyset$
v_5	$\{v_5\} \to \{v_5\}$
v_6	$\{v_5\} \to \{v_5\}$
v_7	$\emptyset \to \emptyset$
v_8	$\{v_5, v_8\} \to \{v_5, v_8\}$
v_9	$\{v_5\} \to \{v_5\}$
v_{10}	$\{v_5, v_8\} \to \{v_5, v_8\}$
v_{11}	$\{v_5\} \to \{v_5\}$

(b) $\mathtt{DL}_{in}$ label update

v	$h(\mathtt{BL}_{in})$
v_1	$\{0\} \to \{0\}$
v_2	$\{1\} \to \{1\}$
v_3	$\{1\} \to \{1\}$
v_4	$\{0\} \to \{0\}$
v_5	$\{1\} \to \{1\}$
v_6	$\{1\} \to \{1\}$
v_7	$\{1\} \to \{1\}$
v_8	$\{0, 1\} \to \{0, 1\}$
v_9	$\{1\} \to \{1\}$
v_{10}	$\{0, 1\} \to \{0, 1\}$
v_{11}	$\{1\} \to \{1\}$

(c) $\mathtt{BL}_{in}$ label update

Fig. 2: Label update for inserting edge (v_9, v_2)

process with pruning. For a visited vertex x, we prune x without traversing $Des(x)$ iff $\mathtt{DL}_{in}(u) \subseteq \mathtt{DL}_{in}(x)$, because all the vertices in $Des(x)$ are deemed to be unaffected as their $\mathtt{DL}_{in}$ labels are supersets of $\mathtt{DL}_{in}(x)$.

Example 3. Figure 2(a) shows an example of edge insertion. Figure 2(b) shows the corresponding $\mathtt{DL}_{in}$ label update process. $\mathtt{DL}_{in}$ label is presented with brackets. Give an edge (v_9, v_2) inserted, $\mathtt{DL}_{in}(v_9)$ is copied to $\mathtt{DL}_{in}(v_2)$. Then an inspection will be processed on $\mathtt{DL}_{in}(v_5)$ and $\mathtt{DL}_{in}(v_6)$. Since $\mathtt{DL}_{in}(v_9)$ is a subset of $\mathtt{DL}_{in}(v_5)$ and $\mathtt{DL}_{in}(v_6)$, vertex v_5 and vertex v_6 are pruned from the BFS. The update progress is then terminated.

DL label only gives positive answer to a reachability query. In poorly connected graphs, DL will degrade to expensive BFS search. Thus, we employ the Bidirectional Leaf (BL) label to complement DL and quickly identify vertex pairs which are not reachable. We omit the update algorithm of BL, as they are very similar to those of DL, except the updates are applied to $\mathtt{BL}_{in}$ and $\mathtt{BL}_{out}$ labels. Figure 2(c) shows the update of $\mathtt{BL}_{in}$ label. Similar to the DL label, the update process will be early terminated as the $\mathtt{BL}_{in}(v_2)$ is totally unaffected after edge insertion. Thus, no $\mathtt{BL}_{in}$ label will be updated in this case.

Update complexity of DBL. In the worst case, all the vertices that reach or are reachable to the updating edges will be visited. Thus, the time complexity of DL and BL is $O((k+k')(m+n))$ where $(m+n)$ is the cost on the 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.

6 Experimental Evaluation

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

Table 2: Dataset statistics

Dataset	$\|V\|$	$\|E\|$	d_{avg}	**Diameter**	**Connectivity** (%)	DAG-$\|V\|$	DAG-$\|E\|$	DAG CONSTRUCT (ms)
LJ	4,847,571	68,993,773	14.23	16	78.9	971,232	1,024,140	2368
Web	875,713	5,105,039	5.83	21	44.0	371,764	517,805	191
Email	265,214	420,045	1.58	14	13.8	231,000	223,004	17
Wiki	2,394,385	5,021,410	2.09	9	26.9	2,281,879	2,311,570	360
BerkStan	685,231	7,600,595	11.09	514	48.8	109,406	583,771	1134
Pokec	1,632,803	30,622,564	18.75	11	80.0	325,892	379,628	86
Twitter	2,881,151	6,439,178	2.23	24	1.9	2,357,437	3,472,200	481
Reddit	2,628,904	57,493,332	21.86	15	69.2	800,001	857,716	1844

6.1 Experimental Setup

Environment: 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.
Datasets: We conduct experiments on 8 real-world datasets (see Table 2). We have collected the following datasets from SNAP [11]. 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. Reddit and Twitter are two social network datasets obtained from [23].

Table 3: Query time (ms) for different landmark nodes selection. A=max($|Pre(\cdot)|, |Suc(\cdot|)$); B=min($|Pre(\cdot)|, |Suc(\cdot|)$); C=$|Pre(\cdot)| + |Suc(\cdot)|$; D is the betweenness centrality; ours=$|Pre(\cdot)| \cdot |Suc(\cdot)|$

Dataset	A	B	C	D	ours
LJ	125.10	127.84	105.88	113.34	108.51
Web	202.16	144.13	142.16	140.79	139.64
Email	37.02	37.01	36.14	38.53	36.38
Wiki	156.21	159.74	153.66	155.45	157.12
Pokec	37.69	64.57	36.96	50.66	34.78
BerkStan	1890	6002	1883	1252	1590
Twitter	719.31	849.78	685.31	727.59	693.71
Reddit	99.21	65.06	62.68	69.62	60.48

6.2 Label Node Selection

DL select the landmark nodes by heuristically approximating the centrality of a vertex u as $M(u) = |Pre(u)| \cdot |Suc(u)|$. Here, we evaluate different heuristic methods for landmark nodes selection. The results are shown in Table 3. 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, $|Pre(\cdot)| + |Suc(\cdot)|$(degree centrality) 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. It needs to mention that, although the betweenness centrality get a medium overall performance, it shows the best performance in BerkStan dataset.

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 as a leaf node. Following the approach for which 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 3 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.

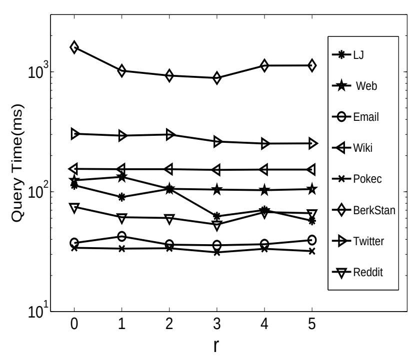


Fig. 3: `BL` label node selection

6.3 Effectiveness of DL+BL

Table 4 shows the percentages of queries answered by `DL` label, `BL` label (*when the other label is disabled*) and `DBL` label. All the queries are randomly generated. The results show that `DL` is effective for dense and highly connected graphs (LJ, Pokec and Reddit) whereas `BL` is effective for sparse and poorly connected graphs (Email, Wiki and Twitter). However, we still incur the expensive `BFS` if the label is disabled. By combining the merits of both indexes, our proposal leads to a significantly better performance. `DBL` could answer much more queries than `DL` and `BL` label. The results have validated our claim that `DL` and `BL` are complementary to each other. We note that the query processing for `DBL` is able

Table 4: Percentages of queries answered by DL label, BL label (when the other label is disabled) and DBL label respectively. We also include the time for DBL to process 1 million queries

Dataset	DL Label	BL Label	DBL Label	DBL time
LJ	97.5%	20.8%	99.8%	108ms
Web	79.5%	54.3%	98.3%	139ms
Email	31.9%	85.4%	99.2%	36ms
Wiki	10.6%	94.3%	99.6%	157ms
Pokec	97.6%	19.9%	99.9%	35ms
BerkStan	87.5%	43.3%	95.0%	1590ms
Twitter	6.6%	94.8%	96.7%	709ms
Reddit	93.7%	30.6%	99.9%	61ms

to handle one million queries with sub-second latency for most datasets, which shows outstanding performance.

Impact of Label Size: On the query processing of DBL. 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 per vertex (by processing 1 million queries) in Table 5.

When varying DL label size k, the performance of most datasets remain stable before a certain size (e.g., 64) 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 incur additional cache misses. BerkStan gets benefit from increasing the DL label size to 128 since 64 landmarks are not enough to cover enough reachable pairs.

Compared with DL label, some of the datasets 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 increasing BL label size and more hash values incorporated, we can quickly prune more unreachable vertex pairs by examining BL label without traversing the graph with BFS. Besides, larger BL size also provides better pruning power of the BFS even if it fails to directly answer the query (Algorithm 2). Nevertheless, the cost of label processing increases with increased BL label size. According to our parameter study, 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.

6.4 General Graph Updates

In this section, we evaluate DBL's performance on general graph update. As DAGGER is the only method that could handle general update, we compare DBL against DAGGER in Figure 4. Ten thousand edge insertion and 1 million queries are randomly generated and performed, respectively. Different from DAGGER, DBL

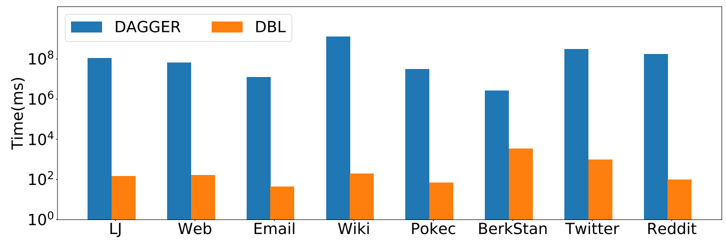


Fig. 4: The execution time for insert 10000 edges as well as 1 million queries

don't need to maintain the DAG, thus, in all the datasets, DBL could achieve great performance lift compared with DAGGER. For both edge insertion and query, DBL is orders of magnitude faster than DAGGER. The minimum performance gap lies in BerkStan. This is because BerkStan has a large diameter. As DBL rely on BFS traversal to update the index. The traversal overheads is crucial for it's performance. BerkStan's diameter is large, it means, during index update, DBL need to traversal extra hops to update the index which will greatly degrade the performance.

6.5 Synthetic graph updates

In this section, we compare our method with IP and TOL. Different from DBL, which could handle real world update, IP and TOL could only handle synthetic edge update that will not trigger DAG maintaining. Thus, for IP and TOL, we follow their experimental setups depict in their paper[27,24]. Specifically, we randomly select 10,000 edges from the DAG and delete them. Then, we will insert the same edges back. In this way, we could get the edge insertion performance without trigger DAG maintenance. For DBL, we stick to general graph updates. The edge insertion will be randomly generated and performed. One million queries will be executed after that. It needs to be noted that, although both IP and TOL claim they can handle dynamic graph, due to their special pre-condition, their methods are in fact of limited use in real world scenario.

(a) Varying BL label sizes

Dataset	16	32	64	128	256
LJ	136.1	131.9	108.1	107.4	110.3
Web	177.2	128.5	152.9	156.6	174.3
Email	77.4	53.9	38.3	41.1	44.4
Wiki	911.6	481.4	273.7	181.3	157.4
Pokec	54.8	43.7	38.6	40.6	53.6
BerkStan	4876.1	4958.9	4862.9	5099.1	5544.3
Twitter	1085.3	845.7	708.2	652.7	673.2
Reddit	117.1	80.4	67.3	63.5	67.9

(b) Varying DL label sizes

Dataset	16	32	64	128	256
LJ	108.2	110.3	106.9	120.2	125.5
Web	154.0	152.5	151.1	158.8	167.8
Email	37.9	39.5	35.8	39.8	43.7
Wiki	274.5	282.6	272.4	274.8	281.1
Pokec	38.1	40.6	36.3	49.7	55.6
BerkStan	6369.8	5853.1	4756.3	1628.3	1735.2
Twitter	716.1	724.4	695.3	707.1	716.9
Reddit	64.6	65.9	62.9	75.4	81.4

Table 5: Query performance(ms) with varying DL and BL label sizes

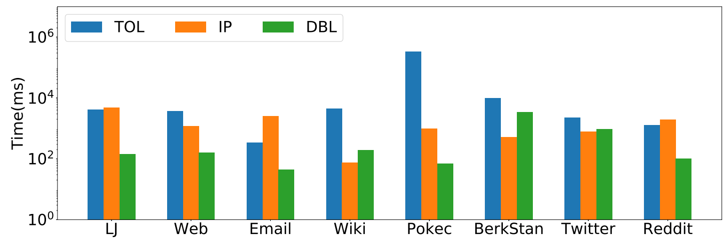


Fig. 5: The execution time for insert 10000 edges as well as 1 million queries, for TOL and IP, the updates are synthetic that will not trigger SCC update

The results are shown in Figure 5. DBL outperforms other baselines in most cases except on three data sets (Wiki, BerkStan and Twitter) where IP could achieve a better performance. Nevertheless, DBL outperforms IP and TOL by 4.4x and 21.2x, respectively with respect to geometric mean performance. We analyze the reason that DBL can be slower than IP on Wiki, BerkStan and Twitter. As we aforementioned, DBL relies on the pruned BFS to update the index, the BFS traversal speed will determine the worst-case update performance. Berkstan has the largest diameter as 514 and Twitter has the second largest diameter as 24, which dramatically degrade the update procedure in DBL. For Wiki, DBL could still achieve a better update performance than IP. However, IP is much more efficiency in query processing which lead to better overall performance.

Although this experimental scenario has been used in previous studies, the comparison is unfair for DBL. As both IP and TOL rely on the DAG to process queries and updates, **their synthetic update exclude the DAG maintaining procedure/overheads from the experiments.** However, DAG maintenance is essential for their method to handle real world edge updates, as we have shown in Figure 4, the overheads is nonnegligible.

6.6 Parallel Performance

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 for query processing. We follow existing GPU-based graph processing pipeline by batching the queries and updates [7,19,12]. Note that the transfer time can be overlapped with GPU processing to minimize data communication costs. Both CPU and GPU implementations are based on the vertex centric framework.

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 6. 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

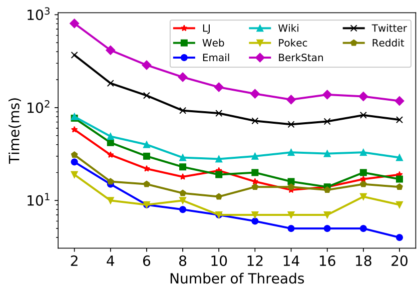


Fig. 6: Scalability of DBL on CPU

Dataset	TOL	IP	IP-P	DBL	DBL-P	DBL-G	B-BFS
LJ	46.6	50.7	24.9	108.1	16.4	**6.1**	555561
Web	40.6	39.7	22.6	139.2	**12.4**	14.2	236892
Email	26.8	21.6	9.4	36.4	4.1	**2.8**	10168
Wiki	74.9	12.7	**4.1**	157.2	28.4	14.8	61113
Pokec	27.2	37.6	23.2	34.8	9.0	**3.1**	253936
BerkStan	37.2	31.6	**16.4**	1590.0	131.0	835.1	598127
Twitter	64.6	30.2	**7.1**	709.1	79.4	202.1	78496
Reddit	56.7	44.7	19.56	61.2	14.6	**3.1**	273935

Table 6: The query performance(ms) on CPU and GPU architectures. B-BFS means the bidirectional BFS

memory bandwidth bound nature of the processing tasks. DBL invokes the BFS traversal once the labels are unable to answer the query and the efficiency of the 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.

The compared query processing performance is shown in Table 6. Bidirectional BFS (B-BFS) query is listed as a baseline. We also compare our parallel solutions with a home-grown OpenMP implementation of IP (denoted as IP-P). Twenty threads are used in the OpenMp implementation. 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 cannot be efficiently parallelized. For our parallel implementation IP-P, we assign a thread to handle one query. We have the following observations.

First, DBL is built on the pruned BFS which can be efficiently parallelized with the vertex-centric paradigm. We have observed significant performance improvement by parallelized executions. DBL-P (CPUs) gets 4x to 10x speedup across all datasets. DBL-G (GPUs) shows an even better performance. In contrast, as DFS incurs frequent random accesses in IP-P, the performance is bounded by memory bandwidth. Thus, parallelization does not bring much performance gain to IP-P compared with its sequential counterpart.

Second, DBL provides competitive efficiency against IP-P but DBL can be slower than TOL and IP when comparing the single thread performance. However, this is achieved by assuming the DAG structure but the DAG-based approaches incur prohibitively high cost of index update, as we demonstrated in the previous subsections. In contrast, DBL achieves sub-second query processing performance for handling 1 million queries while still support efficient updates without using the DAG.

Third, 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, which do not suit the GPU architecture.

7 Conclusion

In this work, we propose DBL, an indexing framework to support dynamic reachability query processing on incremental graphs. To our best knowledge, 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 is 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. 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. As future work, we are interested in extending DBL to support deletions, which will be lazily supported in many applications.

Acknowledgement. Yuchen Li's work was supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No.: MOE2019-T2-2-065).

References

1. Akiba, T., Iwata, Y., Yoshida, Y.: 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. pp. 349–360. ACM (2013)
2. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: Proceedings of the 23rd international conference on World wide web. pp. 237–248 (2014)
3. Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., Hwang, D.U.: Complex networks: Structure and dynamics. Physics reports **424**(4-5), 175–308 (2006)
4. Bramandia, R., Choi, B., Ng, W.K.: Incremental maintenance of 2-hop labeling of large graphs. TKDE **22**(5), 682–698 (2010)
5. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SICOMP **32**(5), 1338–1355 (2003)
6. Demetrescu, C., Italiano, G.F.: Fully dynamic all pairs shortest paths with real edge weights. In: FOCS. pp. 260–267. IEEE (2001)
7. Guo, W., Li, Y., Sha, M., Tan, K.L.: Parallel personalized pagerank on dynamic graphs. PVLDB **11**(1), 93–106 (2017)
8. Henzinger, M.R., King, V.: Fully dynamic biconnectivity and transitive closure. In: FOCS. pp. 664–672. IEEE (1995)
9. Hotz, M., Chondrogiannis, T., Wörteler, L., Grossniklaus, M.: Experiences with implementing landmark embedding in neo4j. In: Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). pp. 1–9 (2019)
10. Jin, R., Ruan, N., Xiang, Y., Wang, H.: Path-tree: An efficient reachability indexing scheme for large directed graphs. TODS **36**(1), 7 (2011)

11. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data` (Jun 2014)
12. Li, Y., Zhu, Q., Lyu, Z., Huang, Z., Sun, J.: Dycuckoo: Dynamic hash tables on gpus. In: ICDE (2021)
13. Potamias, M., Bonchi, F., Castillo, C., Gionis, A.: Fast shortest path distance estimation in large networks. In: CIKM. pp. 867–876. ACM (2009)
14. Qiu, X., Cen, W., Qian, Z., Peng, Y., Zhang, Y., Lin, X., Zhou, J.: Real-time constrained cycle detection in large dynamic graphs. Proceedings of the VLDB Endowment **11**, 1876–1888 (08 2018). https://doi.org/10.14778/3229863.3229874
15. Roditty, L.: Decremental maintenance of strongly connected components. In: SODA. pp. 1143–1150. SIAM (2013)
16. Roditty, L., Zwick, U.: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. SICOMP **45**(3), 712–733 (2016)
17. Schenkel, R., Theobald, A., Weikum, G.: Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In: ICDE. pp. 360–371. IEEE (2005)
18. Seufert, S., Anand, A., Bedathur, S., Weikum, G.: Ferrari: Flexible and efficient reachability range assignment for graph indexing. In: ICDE. pp. 1009–1020. IEEE (2013)
19. Sha, M., Li, Y., He, B., Tan, K.L.: Accelerating dynamic graph analytics on gpus. PVLDB **11**(1) (2017)
20. Su, J., Zhu, Q., Wei, H., Yu, J.X.: Reachability querying: can it be even faster? TKDE (1), 1–1 (2017)
21. Valstar, L.D., Fletcher, G.H., Yoshida, Y.: Landmark indexing for evaluation of label-constrained reachability queries. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 345–358 (2017)
22. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: ICDE. pp. 75–75. IEEE (2006)
23. Wang, Y., Fan, Q., Li, Y., Tan, K.L.: Real-time influence maximization on dynamic social streams. Proceedings of the VLDB Endowment **10**(7), 805–816 (2017)
24. Wei, H., Yu, J.X., Lu, C., Jin, R.: Reachability querying: an independent permutation labeling approach. VLDBJ **27**(1), 1–26 (2018)
25. Yildirim, H., Chaoji, V., Zaki, M.J.: Grail: Scalable reachability index for large graphs. PVLDB **3**(1-2), 276–284 (2010)
26. Yildirim, H., Chaoji, V., Zaki, M.J.: Dagger: A scalable index for reachability queries in large dynamic graphs. arXiv preprint arXiv:1301.0977 (2013)
27. Zhu, A.D., Lin, W., Wang, S., Xiao, X.: Reachability queries on large dynamic graphs: a total order approach. In: SIGMOD. pp. 1323–1334. ACM (2014)