2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining

I/O Efficient Algorithms for Exact Distance Queries on Disk-Resident Dynamic Graphs

Yishi Lin, Xiaowei Chen, John C.S. Lui
Department of Computer Science & Engineering, The Chinese University of Hong Kong
{yslin, xwchen, cslui}@cse.cuhk.edu.hk

***Abstract*—Point-to-point shortest distance queries are fundamental to large graph analytics. Motivated by the need for low-latency distance queries in large-scale "dynamic" graphs, we consider the problem of answering exact shortest distance queries on disk-resident scale-free dynamic graphs. Our query processing uses the *canonical labeling* method, which is a special 2-hop distance labeling for fast distance queries. In this paper, we propose two I/O efficient algorithms to update the *canonical labeling*. To the best of our knowledge, our proposed methods are the first practical disk-based methods to "*incrementally update*" the *canonical labeling* on dynamic graphs. We also show how to answer distance queries on the latest network based on outdated labels and new edges. Extensive experiments demonstrate the efficiency of our methods. Our update methods are an order of magnitude faster than reconstructing the *canonical labeling*. When the number of new edges is small, say less than 1% of the previous number of edges, our query algorithm based on outdated labels provides exact shortest distance and the query time is comparable to other query algorithms using latest labels.**

I. Introduction

Shortest distance computation is fundamental in many large graph analytics, online social network analytics, or computer network applications [1]. For example, to analyze a large graph, we often need to compute closeness centrality, diameter, etc. Many of these applications require efficient computation of shortest distances. For instance, in online social networks (OSNs), users may request a list of users sharing common interests. The efficiency of recommendation algorithms based on shortest distances also relies on fast distance query.

To substantially reduce the latency for answering distance queries, a series of works focused on building space-efficient data structures for fast distance query. In the seminal work [2], Cohen et al. first proposed the idea of 2*-hop labeling*, a data structure for answering *Point-to-Point* (P2P) shortest distance. Cohen et al. proved that finding the 2-hop labeling with minimum size is NP-hard. Researchers have proposed various methods to efficiently construct small-sized 2-hop labeling. Some recent works include [3], [4], [5], [6], [7]. Akiba et al. proposed the *Pruned Landmark Labeling* (PLL) [5], and an incrementally update method *Dynamic PLL* [6]. Jiang et al. proposed the disk-based *Hop-Doubling Labeling* method for static scale-free networks [7]. To the best of our knowledge, *Dynamic PLL* is the state-of-the-art method to efficiently process incremental graph updates, and *Hop-Doubling Labeling* is the state-of-the-art method for disk-based static graphs. Note that PLL and *Hop-Doubling Labeling* are essentially special cases of the *canonical labeling* formally defined in [3].

Challenges: In this work, we consider how to efficiently answer exact distance queries on disk-resident graphs which are both *dynamic* and *scale-free*. We use the *canonical labeling* to answer distance queries. We summarize the challenges as follows. First, to answer exact distance queries, how can we efficiently update the *canonical labeling* of the graph? Second, the graph may change quickly, how can we answer exact distance queries before the update finishes? The issue is how we can design an I/O efficient algorithm to update the labeling and at the same time, quickly answer exact distance queries.

Contributions: To the best of our knowledge, there is no existing work that could efficiently answer distance queries on disk-resident dynamic graphs via 2-hop labeling. For example, the *Dynamic PLL* [6] cannot handle disk-resident graphs, and the work proposing *Hop-Doubling Labeling* [7] does not consider dynamic graphs. In this paper, our contributions are:

- We propose a *single edge update* algorithm that processes each inserted edge, and efficiently updates the *canonical labeling*. We also present several refinements which further speed up the *single edge update* method.
- We propose a *batch update* algorithm which takes a batch of inserted edges as input, and update the *canonical labeling*.
- We propose a method to answer distance queries toward the most updated graph using only outdated *canonical labeling* and a set of new edges. This is useful when we decide not to update the labeling stored on disk, or when we are in the progress of updating the *canonical labeling*.
- We conduct extensive experiments using real scale-free networks. Experimental results show that both of our update methods could run up to an order of magnitude faster than reconstructing the labels. And, given that the number of new edges is not too large, the performance of our query algorithm based on out-dated labels is comparable with that of query algorithm based on latest labels.

This is the outline of our paper. Section II gives preliminaries. We present the *single edge update* method in Section III, the *batch update* method in Section IV, and the *query* algorithm based on outdated labels in Secion V. We show experimental results in Section VI. Section VII concludes.

II. Preliminaries

A. Notations

Static graph: A static network can be modeled as a graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ directed edges. For a node $v \in V$, we denote its out-neighbor (resp. in-neighbor) by $N^+(v)$ (resp. $N^-(v)$). For two nodes u and v, we denote the distance of the shortest path from u to v by $d(u, v)$. If u cannot reach v, let $d(u, v) = +\infty$. We focus on unweighed graphs (or networks) in this paper.

Dynamic graph: For a network, we denote its snapshot at time

ASONAM '15, August 25-28, 2015, Paris, France

DOI: http://dx.doi.org/10.1145/2808797.2808876

t by $G_t = (V_t, E_t)$. For simplicity, we assume that timestamps are contiguous integers and G_0 is the initial snapshot of the network. Because we only consider the insertion of nodes and edges, we assume $V_{t-1} \subseteq V_t$ and $E_{t-1} \subseteq E_t$ for all $t > 0$. We sometimes use G to denote the *latest snapshot* of the network and let $d(u, v)$ be the latest distance from u to v.

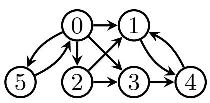
Problem definition: In this paper, given snapshots of a dynamic network, we focus on how to efficiently answer point-to-point (P2P) distance queries on the latest snapshot. To be more specific, our goal is to provide I/O efficient algorithms to answer queries about $d(u, v)$, for $u, v \in V$.

Computation model: The analysis of our disk-based algorithm is based on the *external memory model* [8]. Let M denote the memory size and B denote the disk block size. We assume $1 \leq B \ll M$. The I/O cost of reading or writing N elements is denoted by $scan(N) = \Theta(N/B)$. The I/O cost for sorting N elements is denoted by $sort(N) = O(\frac{N}{B} \log_{M/B} \frac{N}{B})$.

B. 2-Hop Labeling

We first introduce the general idea of answering distance query for "*static graphs*" via *2-hop labeling* [2]. For a given graph G, we first pre-process it and construct the *2-hop labeling* for it. The *2-hop labeling* is defined as follows.

Definition 1 (2-hop Labeling)**.** *In a 2-hop labeling, every node $u \in V$ has an out-label $L_{out}(u)$ and an in-label $L_{in}(u)$. Label $L_{out}(u)$ contains a set of out-entries in the form of (v, d), which denotes the shortest distance from u to v is d. Similarly, label $L_{in}(u)$ contains a set of in-entries in the form of (v, d), which denotes the shortest distance from v to u is d. If node x can reach node y, we require that there exist $(w, d_1) \in L_{out}(x)$ and $(w, d_2) \in L_{in}(y)$ such that $d(x, y) = d_1 + d_2$. We say that the pair (x, y) is **covered** by w, or the (distance of) node pair (x, y) is **covered** by entries $(w, d_1) \in L_{out}(x)$ and $(w, d_2) \in L_{in}(y)$. The set of labels for all nodes is denoted by L.*



Node	L_{out}	L_{in}
0	$(0,0)$	$(0,0)$
1	$(1,0)$	$(0,1),(1,0)$
2	$(1,1),(2,0)$	$(0,1),(2,0)$
3	$(1,2),(3,0)$	$(0,1),(2,1),(3,0)$
4	$(1,1),(4,0)$	$(0,2),(1,1),(3,1),(4,0)$
5	$(0,1),(5,0)$	$(0,1),(5,0)$

Fig. 1. A simple graph and one of its 2-hop labelings.

To illustrate, consider the static graph G and one of its 2-hop labelings in Figure 1. Once we have a 2-hop labeling, it is easy to see that we could answer distance query from node u to v using only $L_{\text{out}}(u)$ and $L_{\text{in}}(v)$. Note that there are trivial entries $(u, 0) \in L_{\text{out}}(u)$ and $(v, 0) \in L_{\text{out}}(v)$. With trivial entries, we could write the query processing in a compact form:

$$\texttt{QUERY}(L,u,v) = \min\{d_1+d_2 \mid (w,d_1) \in L_{\text{out}}(u), (w,d_2) \in L_{\text{in}}(v)\}.$$

To simplify the notations, we also denote an out-entry $(v, d) \in L_{\text{out}}(u)$ by $(\underline{u} \to v, d) \in L_{\text{out}}$ and an in-entry $(v, d) \in L_{\text{in}}(u)$ by $(v \to \underline{u}, d) \in L_{\text{in}}$. The size of a 2-hop labeling is defined as $\sum_{v \in V}(|L_{\text{out}}(v)| + |L_{\text{in}}(v)|)$. Constructing a small-sized 2-hop labeling is the key to the query efficiency.

Abraham et al. formally defined a special type of 2-hop labeling named the *canonical labeling* [3]:

Definition 2 (Canonical Labeling)**.** *We are given a ranking r of nodes in V where each node has a unique rank. For nodes $u, v \in V$, let handle(u, v) denote the node with the highest rank among nodes in all shortest path from u to v. In the canonical labeling, label $L_{out}(u)$ contains (v, d) if and only if handle$(u, v) = v$ and $d(u, v) = d$. Label $L_{in}(v)$ contains (u, d) if and only if handle$(u, v) = u$ and $d(u, v) = d$.*

The definition of the *canonical labeling* guarantees that all distance queries could be answered correctly. Also, the labeling is minimal, meaning that every entry is necessary for answering some distance queries. Suppose in Figure 1, nodes with smaller id have higher rank, then, the labeling in Figure 1 is a *canonical labeling* based on the rank.

C. Hop-Doubling Labeling

Recently, Jiang et al. proposed a *Hop-Doubling Labeling* technique for answering distance queries on unweighed scale-free "*static*" networks. They proposed an I/O efficient labeling algorithm for disk-resident graphs. Since we are interested in extending this work to answer distance queries for disk-based scale-free dynamic graphs, we first briefly review the main results in [7]. The *Hop-Doubling Labeling* algorithm first ranks nodes according to the "*product*" of their in-degree and out-degree, where the highest rank is given to the node with the largest product. Then, the algorithm iteratively generates label entries to cover node pairs with increasing distances. Let r denote the ranking of nodes, the *Hop-Doubling Labeling* algorithm generates the *canonical labeling* based on r.

Jiang et al.[7] proved that given an unweighted scale-free static graph G, the number of entries generated for any vertex is $O(h)$, where h is assumed to be a small integer based on the properties of scale-free networks. The intuition behind the small label size is as follows. We say a path p is *hit* by a node v, if p passes through it. Intuitively, for a scale-free network, a significantly large fraction of long shortest paths could be hit by a small number of top degree nodes. The *Hop-Doubling Labeling* algorithm tries to place high degree nodes into labels of relevant nodes, so that a small number of label entries could cover a large fraction of distance queries. Moreover, for every node u, there exists a small set of nodes in its neighborhood so that all short shortest paths passing through u could be hit by these nodes. The *Hop-Doubling Labeling* method determines a very small number of label entries to cover queries about node pairs with small distance.

D. Incremental Maintenance Objective

In reality, many real-networks (e.g., online social networks) are growing. In order to efficiently answer distance queries in the latest network G, we need a 2-hop labeling of G. One straightforward strategy is to reconstruct the 2-hop labeling based on the latest graph G. However, for disk-based graphs, it is extremely time consuming even with the state-of-the-art method. For example, in our experiments, the *Hop-Doubling Labeling* algorithm [7] takes almost three hours to construct a 2-hop labeling for a graph with 3.2 million nodes and 9.4 million edges. Another problem is that one cannot answer distance queries efficiently until the construction finishes.

In this paper, we propose two methods to incrementally update the canonical labeling, each has its own merits. Suppose we have a *canonical labeling* L^{t-1} for graph G_{t-1} based on rank r. Unless we state otherwise, we assume that $|L_{\text{in}}^{t-1}(u)|$ and $|L_{\text{out}}^{t-1}(u)|$ are small for all $u \in V$. The labeling L^{t-1}

generated by the *Hop-Doubling Labeling* method is one example. Our goal is to update L^{t-1} and obtain an r-based *canonical labeling* for the latest graph. Note that we reuse the rank r computed in past snapshots for good reasons. First, while a scale-free network grows, high degree nodes (high rank nodes) tend to receive more links, which is justified by the *preferentially attachment* [9]. Hence, it is safe to assume that high rank nodes in past snapshots tend to have high rank in the latest snapshot. And, these high rank nodes still hit a great fraction of long shortest paths in the latest network. Thus, in the updated labeling based on r, the label size for every node is still small. Second, it helps us to keep the labeling minimal. We reuse rank r in both of our update methods, and our experiments show only a small increase in the label size.

III. Single Edge Update for Dynamic Networks

In this section, we consider how to efficiently update *canonical labeling* when a new edge is inserted into the graph. Let G_{t-1} be the old graph and L^{t-1} be its corresponding label. Suppose L^{t-1} is *based on* the rank r. We use G to denote the new graph with the new edge e_{xy}. Without loss of generality, we assume $x \neq y$ and $e_{xy} \notin E_{t-1}$. The update method has two phases, the *patch generation phase* and the *patch merge phase*. In the *patch generation phase*, we aim to create a minimal set of supplementary labels so that we could correctly answer all distance queries. Let P be the set of newly created entries, we refer to it as the *patch* of labels. Patch P could be kept in memory or written to disk. In the *patch merge phase*, we merge patch labels P with L^{t-1} and obtain L. In order to keep the labeling L minimal, we also remove entries in L^{t-1} which are no longer necessary for any distance query.

A. Patch Generation

In the *patch generation* phase, we create a minimal set P of label entries to reflect the graph update. A key fact for the distance change after an edge addition is that, for nodes u and v, $d(u,v) < d_{t-1}(u,v)$ if and only if every shortest path from u to v in G passes through the new edge e_{xy}. Let *Source*$(e_{xy}) = \{u|d(u,y) < d_{t-1}(u,y)\}$ be the set of nodes whose distances to y decrease after the insertion of e_{xy}.

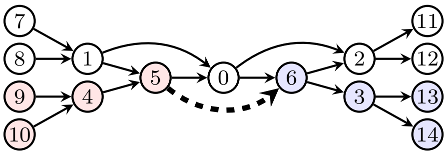


Fig. 2. Example of "sink" and "source": The graph G_{t-1} contains 15 nodes and 16 edges each represented by a solid arrow. We insert an edge e_{56} into G_{t-1} and obtain G. We have *Source*$(e_{56}) = \{4,5,9,10\}$ and *Sink*$(e_{56}) = \{3,6,13,14\}$.

Similarly, let *Sink*$(e_{xy}) = \{u|d(x,u) < d_{t-1}(x,u)\}$. The distance from x to every node in *Sink*(e_{xy}) decreases after the insertion of e_{xy}. Then, $d(u,v) < d_{t-1}(u,v)$ if and only if $u \in$ *Source*(e_{xy}) and $v \in$ *Sink*(e_{xy}). Moreover, $\forall u \in$ *Source*$(e_{xy}), v \in$ *Sink*(e_{xy}), we know $d(u,v) = d_{t-1}(u,x)+1+d_{t-1}(y,v)$. Figure 2 shows an example of "sink" and "source". Our objective of the *patch generation* phase is as follows.

Objective 1 (Patch Generation Objective)**.** *Suppose $u \in$ Source(e_{xy}) and $v \in$ Sink(e_{xy}). Let $w =$ handle(u,v). If $(w, d(u,w)) \notin L_{out}^{t-1}(u)$, then $(w, d(u,w)) \in P_{out}(u)$. Similarly, if $(w, d(w,v)) \notin L_{in}^{t-1}(v)$, then $(w, d(w,v)) \in P_{in}(v)$.*

With a patch P achieving the above objective, for any nodes u and v such that $d(u,v) < +\infty$, the shortest distance between them is *covered* by *handle*(u,v). Therefore, for all $u, v \in V$, we have $d(u,v) = \mathtt{QUERY}(L^{t-1} \cup P) = \min\{d_1 + d_2 | (w,d_1) \in L_{\text{out}}^{t-1}(u) \cup P_{\text{out}}(u), (w,d_2) \in L_{\text{in}}^{t-1}(v) \cup P_{\text{in}}(v)\}$. We now provide a lemma describing how to efficiently find *handles* that should be placed into P.

Lemma 1. *Suppose $u \in$ Source(e_{xy}), $v \in$ Sink(e_{xy}), and $w =$ handle(u,v). We have $(w, d_{t-1}(w,x)) \in L_{in}^{t-1}(x)$ and/or $(w, d_{t-1}(y,w)) \in L_{out}^{t-1}(y)$.*

Proof: If $w = x$ or $w = y$, the lemma holds naturally according to the definition of the *canonical labeling*. Now, assume $w \neq x$ and $w \neq y$. From $w =$ *handle*(u,v) and the fact that every shortest path from u to v now passes through e_{xy}, we can conclude that $w =$ *handle*(u,x) and/or $w =$ *handle*(y,v). Moreover, because none of the shortest paths from u to x passes through e_{xy} in G, we know $d_{t-1}(u,x) = d(u,x)$ and *handle*$(u,x) =$ *handle*$_{t-1}(u,x)$. Similarly, *handle*$(y,v) =$ *handle*$_{t-1}(y,v)$ holds. From the definition of the *handle*, we have $(w, d_{t-1}(w,x)) \in L_{\text{in}}^{t-1}(x)$ if $w =$ *handle*$_{t-1}(u,x)$, and we have $(w, d_{t-1}(y,w)) \in L_{\text{out}}^{t-1}(y)$ if $w =$ *handle*$_{t-1}(y,v)$. ■

With a slight abuse of notation, we also consider $L_{\text{in}}^{t-1}(x)$ and $L_{\text{out}}^{t-1}(y)$ as subsets of nodes in V. From Lemma 1, for any $u \in$ *Source*(e_{xy}) and $v \in$ *Sink*(e_{xy}), set $L_{\text{in}}^{t-1}(x) \cup L_{\text{out}}^{t-1}(y)$ contains *handle*(u,v). The key idea for generating entries in P is that, in order to correctly answer all distance queries from a node in *Source*(e_{xy}) to a node in *Sink*(e_{xy}), we need to place entries containing relevant handles to relevant nodes labels. Moreover, our goal is to keep P minimal. We now describe the rules to generate patch entries. For $u \in$ *Source*(e_{xy}), we insert $(\underline{u} \rightarrow w, d(u,w))$ into P if $w =$ *handle*(u,w), $d(u,w) < d_{t-1}(u,w)$. Such a node w must be in $L_{\text{out}}^{t-1}(y)$. For $v \in$ *Sink*(e_{xy}), we insert $(w \rightarrow \underline{v}, d(w,v))$ into P if $w =$ *handle*(w,v), $d(w,v) < d_{t-1}(w,v)$. Node w must be in $L_{\text{in}}^{t-1}(x)$. To illustrate, consider Figure 2. The patch entries are $P_{\text{in}} = \{(5 \rightarrow \underline{6}, 2), (4 \rightarrow \underline{6}, 3)\}$ and $P_{\text{out}} = \{(\underline{5} \rightarrow 3, 2), (\underline{4} \rightarrow 3, 3), (\underline{9} \rightarrow 3, 4), (\underline{10} \rightarrow 3, 4)\}$. The following corollary holds from Lemma 1.

Corollary 1. *Set P achieves Objective 1 so we could answer arbitrary distance query correctly with $L^{t-1} \cup P$. Moreover, $L^{t-1} \cup P$ is a superset of the canonical labeling based on r.*

Now we provide a lemma showing that set P is minimal.

Lemma 2. *The patch P is minimal, i.e., one will not correctly answer some distance queries if we remove any entry in P.*

Proof: We first study entries in P_{in}. Suppose P_{in} contains entry $(w \rightarrow \underline{v}, d(w,v))$, we know $w =$ *handle*(w,v). If we remove this entry, we cannot answer distance query about $d(w,v)$ correctly. We show this by contradiction. Suppose we are able to compute $d(w,v)$ without entry $(w \rightarrow \underline{v}, d(w,v))$, there must exist a node $u \neq w$ such that $(u, d(w,u)) \in L_{\text{out}}^{t-1}(w) \cup P_{\text{out}}(w)$, $(u, d(u,v)) \in L_{\text{in}}^{t-1}(v) \cup P_{\text{in}}(v)$ and $d(w,v) = d(w,u)+d(u,v)$. Then, $r(u) > r(w)$ holds because $u \neq w$. This implies that node w does not have the highest rank among the set of nodes in one of the shortest paths from w to v, which contradicts with the fact that $w =$ *handle*(w,v). Therefore, every entry in P_{in} is necessary for at least one distance query. Similarly, entries in P_{out} are also minimal. ■

Algorithm 1 shows the pseudo code for generating patch entries in P_{in}. Line 2-12 run a Breadth-First Search (BFS) to place handles in $L_{\text{in}}^{t-1}(x)$ to labels of relevant nodes. In BFS, only (a subset of) nodes in $\mathit{Sink}(e_{xy})$ will be pushed into the queue. For every node u dequeued, Line 4-7 insert entry $(w \rightarrow \underline{u}, d(w,u))$ into $P_{\text{in}}(u)$ for every $w \in L_{\text{in}}^{t-1}(x)$ satisfying $d(w,u) < d_{t-1}(w,u)$ and $w = \mathit{handle}(w,u)$. Note that $r(\mathit{handle}(x,u))$ is essentially the highest rank among all shortest paths from x to u, the condition $r(w) \geq r(\mathit{handle}(x,u))$ in Line 6 is equivalent with condition $w = \mathit{handle}(w,u)$. The value of $r(\mathit{handle}(x,u))$ for node u visited is easy to maintain during the BFS. Line 11 guarantees only nodes in $\mathit{Sink}(e_{xy})$ will be pushed into the queue. One key feature of our patch generation method is that it may not push all nodes in $\mathit{Sink}(e_{xy})$ into queue. Line 7-9 effectively prune the BFS. Suppose there is a node $u \in \mathit{Sink}(e_{xy})$ such that $P_{\text{in}}(u) = \emptyset$, no in-entry will be added to any descent of u in the BFS tree. In this case, the pruning strategy effectively prunes the BFS at node u. For example, in Figure 2, we would have $P_{\text{in}}(3) = \emptyset$ and there is no need to visit "sink" nodes 13 and 14. The generation of P_{out} is symmetric, so we omit the pseudo code.

Algorithm 1 Patch Generation (P_{in})

Input: G_{t-1}, L^{t-1}, r, e_{xy}
Output: P_{in} (patch entries)

▷ *initialize P_{in}*
1: $P_{\text{in}}(u) \leftarrow \emptyset, \forall u \in V$
▷ *run pruned BFS to generate patch entries*
2: $Q \leftarrow$ a queue with element $(y, d_{xy} = 1)$
3: **while** Q is not empty **do**
4: dequeue (u, d_{xu}) from Q
▷ *generate patch entries*
5: **for all** $(w, d(w,x)) \in L_{\text{in}}^{t-1}(x)$ **do**
6: **if** $r(w) \geq r(\mathit{handle}(x,u))$ and QUERY$(L^{t-1}, w, u) > d(w,x) + d_{xu}$ **then**
7: $P_{\text{in}}(u) \leftarrow P_{\text{in}}(u) \cup \{(w, d(w,x) + d_{xu})\}$
▷ *try to prune the BFS*
8: **if** $P_{\text{in}}(u) = \emptyset$ **then**
9: prune the BFS and continue to dequeue
▷ *visit neighbors of u*
10: **for all** unvisited out-neighbor v of u **do**
11: **if** QUERY$(L^{t-1}, x, v) > d_{xu} + 1$ **then**
12: enqueue $(v, d_{xv} = d_{xu} + 1)$ onto Q

Analysis of Algorithm 1: We analyze the I/O cost for generating the patch. We make the following mild and realistic assumptions. First, while generating P_{in} (resp. P_{out}), we assume that $L_{\text{out}}^{t-1}(x)$ and $L_{\text{out}}^{t-1}(w), \forall w \in L_{\text{in}}^{t-1}(x)$ (resp. $L_{\text{in}}^{t-1}(y)$ and $L_{\text{in}}^{t-1}(w), \forall w \in L_{\text{out}}^{t-1}(y)$) fits into the main memory. We also assume loading $L_{\text{in}}(y)$ or $L_{\text{out}}(u)$ of a node u from disk costs $O(1)$ I/Os. These two assumptions are realistic because we have assumed that the labeling is based on a rank r so that the number of entries for every node is small. We also assume loading in-edges or out-edges of a given node costs $O(1)$ I/Os since real networks are sparse. The I/O cost of the *patch generation* phase is stated as follows.

Lemma 3. *Let e_{xy} be the newly inserted edge, define $\mathit{Sink}^{+}(e_{xy}) = \mathit{Sink}(e_{xy}) \cup \{u | u \in N^{+}(v), v \in \mathit{Sink}(e_{xy})\}$, and $\mathit{Source}^{-}(e_{xy}) = \mathit{Source}(e_{xy}) \cup \{u | u \in N^{-}(v), v \in \mathit{Source}(e_{xy})\}$. The patch generation phase performs $O(|\mathit{Sink}^{+}(e_{xy})| + |\mathit{Source}^{-}(e_{xy})|)$ I/Os in the worst case. We need to perform $scan(|P|) = \Theta(|P|/B)$ extra I/Os if we write P to disk.*

Proof: We first analyze Algorithm 1 which generates P_{in}. For every node u visited in BFS, Line 2-12 load its out-edges at most once and $L_{\text{in}}^{t-1}(u)$ at most twice. In the worst case where no node is pruned by Line 7-9, the number of nodes visited by BFS is $|\mathit{Sink}^{+}(e_{xy})|$. Therefore, the worst case I/O cost is $O(|\mathit{Sink}^{+}(e_{xy})|)$. Similarly, the I/O cost for generating P_{out} is $O(|\mathit{Source}^{-}(e_{xy})|)$ in the worst case. ■

Note that in practice, the pruning strategy in Line 7-9 is effective and the actual performance is much better than the $O(|\mathit{Sink}^{+}(e_{xy})| + |\mathit{Source}^{-}(e_{xy})|)$ bound.

B. Patch Merge

In the *patch merge* phase, we first prune label entries that are no longer necessary for answering distance queries. Then, we merge labels survived from the pruning and obtain the updated label L. We use the following *pruning rule*: we remove an entry $(u \rightarrow v, d)$ if there exist $(\underline{u} \rightarrow w, d_1)$ and $(w \rightarrow \underline{v}, d_2)$ so that $d_1 > 0$, $d_2 > 0$ and $d_1 + d_2 \leq d$. Algorithm 2 depicts the pseudo code for the *patch merge* phase for in-labels. Note that P is minimal, there is no need to try to prune patch entries.

Algorithm 2 Patch Merge (L_{in})

Input: L^{t-1} on disk, P on disk
Output: L_{in} on disk

▷ *L_{in}^{t-1}, P_{in} contains $(w \rightarrow \underline{v}, d)$ sorted by v*
▷ *L_{out}^{t-1}, P_{out} contains $(\underline{u} \rightarrow w, d)$ sorted by u*
allocate Buf_{in} with size $M - 2B$ to load $L_{\text{in}}(u)$ and $P_{\text{in}}(u)$ in batch
allocate Buf_{out} with size B to load $L_{\text{out}}(u)$ and $P_{\text{out}}(u)$ in batch
for each buffer block Buf_{in} **do**
set the status of all label entries in Buf_{in} to *unpruned*
for each buffer block Buf_{out} **do**
for each unpruned entry $(u \rightarrow \underline{v}, d)$ in Buf_{in} **do**
for each entry $(\underline{u} \rightarrow w, d_1)$ $(d_1 > 0)$ in Buf_{out} **do**
if $(w \rightarrow \underline{v}, d_2) \in \text{Buf}_{\text{in}}, d_2 > 0, d_1 + d_2 \leq d$ **then**
set the status of $(u \rightarrow \underline{v}, d)$ to *pruned*
write all unpruned entries in Buf_{in} to disk

Analysis of Algorithm 2: We allocate buffer Buf_{in} with size $M - 2B$ and buffer Buf_{out} with B while pruning in-entries. The last block of memory serves as the output buffer. Then, we load out-entries into memory $\lceil (|L_{\text{in}}^{t-1}| + |P_{\text{in}}|)/(M - 2B) \rceil$ times. We load every in-entries into memory once. The amount of data we write to disk is $|L_{\text{in}}|$, which is at most $|L_{\text{in}}^{t-1}| + |P_{\text{in}}|$. Therefore, the I/O cost to obtain L_{in} is $O(\lceil (|L_{\text{in}}^{t-1}| + |P_{\text{in}}|)/M \rceil \cdot scan(|L_{\text{out}}^{t-1}| + |P_{\text{out}}|) + scan(|L_{\text{in}}^{t-1}| + |P_{\text{in}}|))$. The analysis of the I/O cost to obtain L_{out} is similar. We summarize the total I/O cost to obtain L from P and L^{t-1} in the following lemma.

Lemma 4. *The total I/O cost for the patch merge phase is*

$$O\left(\lceil (|L^{t-1}| + |P|)/M \rceil \cdot scan(|L^{t-1}| + |P|) \right).$$

Note that our buffer allocation strategy differs from that in the pruning of the *Hop-Doubling Labeling* algorithm [7], which allocates two buffers, each with size $M/2$. To obtain L_{in}, they load out-entries into memory $\lceil (2|L_{\text{in}}^{t-1}| + |P_{\text{in}}|)/M \rceil$ times. Our method only loads out-entries into memory $\lceil (|L_{\text{in}}^{t-1}| + |P_{\text{in}}|)/(M - 2B) \rceil$ times.

From Corollary 1, Lemma 3 and Lemma 4, we have the following theorem for the *single edge update* method.

Theorem 1 (Single Edge Update)**.** *The I/O complexity for the single edge update method is*

$$O\Big(|\mathit{Sink}^{+}(e_{xy})|+|\mathit{Source}^{-}(e_{xy})|+\Big\lceil\frac{|L^{t-1}|+|P|}{M}\Big\rceil\cdot scan\left(|L^{t-1}|+|P|\right)\Big).$$

The new labeling L is a canonical labeling based on r.

C. Refinements

Lazy Patch Merge. Note that the *patch merge* phase is time-consuming compared with the *patch generation* phase. There are two main reasons. First, even if P is almost empty and we do not prune label entries, loading L^{t-1} from disk and writing L to disk require performing $O(|L^{t-1}|/B)$ I/Os. Second, in order to prune unnecessary label entries, we need to examine every entry in L^{t-1} and see whether it could be removed. The examination process has high I/O cost as one can see from Algorithm 2. For these two practical concerns, we recommend performing *lazy patch merge* as follows. Suppose the initial graph is $G_0 = (V_0, E_0)$ and $E_t/E_{t-1} = \{e_t\}$ for all $t \geq 1$. Given L^0 and e_1, we perform a single edge update without the merge phase and obtain a set P^1 of patch entries. We keep P^1 in memory. Given $L^0 \cup P^1$ and a new edge e_2, we generate patch entries P^2. Likewise, we generate patches for new edges one by one, until the size of all patches exceeds a pre-specified threshold. Suppose the size of patches exceeds our threshold after generating patch entries for e_t, we write $P = P^1 \cup P^2 \cup \cdots \cup P^t$ to disk. Then, we perform *patch merge* as described in Algorithm 2 to prune unnecessary entries and obtain L^t. Note that we also prune entries in P^i for all $i < t$, because some entries there might be unnecessary after the insertion of newer edges. After the *lazy patch merge* process, we proceed to process the insertion of e_{t+1}.

Label Prefetch. To reduce the number of I/Os during the patch generation phase, we allocate a small buffer to keep some *"hot data"* (i.e., label) in memory. Recall that to generate P when there is a new edge e_{xy}, we need to load $L_{\text{out}}^{t-1}(w_1), \forall w_1 \in L_{\text{in}}^{t-1}(x)$ and $L_{\text{in}}^{t-1}(w_2), \forall w_2 \in L_{\text{out}}^{t-1}(y)$. For a *canonical labeling* where nodes with top ranks tend to appear in labels of a large fraction of nodes, we know that $L^{t-1}(u)$ tends to be "hot" if u has a high rank. Therefore, we load labels of nodes in decreasing order of their ranks into main memory until the buffer is full. We will show via experiments the impact of the buffer size using real data.

IV. Batch Update for Dynamic Networks

We now show how to efficiently update the *degree-based canonical labeling* when a set of new edges is inserted. One way is to use the *single edge update* method to handle the insertion of each new edge. However, the I/O cost of repeatedly applying the single edge update method grows almost linear with the number of new edges. Here, we propose the *batch update* method, which has smaller I/O cost compared with repeatedly applying the *single edge update* when the number of new edges is very large. We assume that we have a *canonical labeling* L^{t-1} for graph G_{t-1} and L^{t-1} is based on rank r. Our goal is to generate the *canonical labeling* based on r for the latest graph G. To simplify notations, we assume $V = V_{t-1}$ and isolated nodes in V_{t-1} have the lowest ranks in r.

A. Basic Idea

The batch update method iteratively generates entries in the *canonical labeling*. Moreover, it utilizes entries in L^{t-1} so as to avoid generating label entries from the scratch. In each iteration, we denote the set of entries generated in the current iteration as L_{cand}. We treat the initialization process as the 0-th iteration. Before initialization, let $L = L^{t-1}$. In the 0-th iteration, L_{cand} is constructed as follows.

$$\begin{aligned} L_{\text{cand}} = &\{(\underline{u} \to v, 1) | e_{uv} \in E \backslash E_{t-1}, r(u) > r(v)\}, \\ &\cup \{(u \to \underline{v}, 1) | e_{uv} \in E \backslash E_{t-1}, r(v) > r(u)\}. \end{aligned} \quad (1)$$

Note that our goal is to let L be an r-based *canonical labeling*, which is minimal. Hence, at the end of each iteration, including the special initialization iteration, we merge L_{cand} into L using the same method as the *patch merge* phase for the *single edge update*. The set L_{cand} of entries can be treated as a set of *patch* entries for L.

Denote the set of entries in an r-based *canonical labeling* with distance d by C_d, where $d \geq 0$. Let D_G be the diameter of graph G, it is easy to see $C_d = \emptyset, \forall d > D_G$. From Eq. (1), we have the following lemma about the initialization.

Lemma 5. *After the initialization, we have $C_0 \cup C_1 \subseteq L$.*

The high level idea of the batch update methods is that we expand L by inserting entries in $C_2, C_3, \ldots, C_{D_G}$ into L iteratively. Also, to keep L minimal, we prune entries that are no longer necessary for distance queries.

In the i-th iteration ($i \geq 1$), denote the set of entries generated and survived from the pruning in the $(i-1)$-th iteration by L_{prev}. Label L contains entries generated and survived from the pruning in all previous iterations. We rewrite the generation rules [7] in the following compact form. In the i-th iteration, we generate entries in L_{cand} as follows.

1) Suppose $(u \to v, d_1) \in L$ and $(\underline{v} \to w, d_2) \in L_{\text{prev}}$, we insert $(\underline{u} \to w, d_1 + d_2)$ into L_{cand} if $r(w) > r(u)$ and there does not exist $(\underline{u} \to w, d) \in L$ such that $d \leq d_1 + d_2$.
2) Suppose $(u \to v, d_1) \in L$ and $(w \to \underline{u}, d_2) \in L_{\text{prev}}$, we insert $(w \to \underline{v}, d_1 + d_2)$ into L_{cand} if $r(w) > r(v)$ and there does not exist $(w \to \underline{v}, d) \in L$ such that $d \leq d_1 + d_2$.

Note that the existence of entry $(u \to v, d)$ implies that there is a path from u to v with distance d. The intuition behind generation rules is that we try to concatenate two paths into a longer one. When we finish generating L_{cand}, we merge it into L using the same method as the *patch merge* phase for the *single edge update* method. To be specific, we prune every entry in L_{cand} and L, and merge entries survived from the pruning into the new L.

The batch update method terminates naturally when $L_{\text{cand}} = \emptyset$ after the pruning. Note that if $E_{t-1} = \emptyset$, the *batch update* method degenerates into the *Hop-Doubling Labeling* algorithm. The following lemma for the *Hop-Doubling Labeling* algorithm also holds for the *batch udpate* method.

Lemma 6. *After the $2i$-th iteration ($0 \leq i \leq \lceil \log(D_G) \rceil$), we have $C_d \subseteq L$ for all $d \leq 2^i$.*

For the proof of Lemma 6, we refer interested readers to the proof by Jiang et al. [7]. From Lemma 6, we have the following theorem about the *batch update* method.

Theorem 2. *The batch update method returns a canonical labeling L based on rank r.*

Proof: From Lemma 6, we know L is a superset of an r-based canonical labeling. Moreover, L is minimal because every entry survives from the pruning. Thus, we can conclude that L is an r-based canonical labeling. ∎

I/O efficient candidate generation. We describe our method to generate entries in L_{cand}. In each iteration, we allocate buffer Buf_{all} with size $M-2B$ to load entries in L and we allocate buffer Buf_{prev} with size B to load entries in L_{prev}. We adopt a similar nested loop join strategy as the *patch merge* algorithm. We generate out-entries in L_{cand} as follows. We first sort entries $(u \to \underline{v}, d)$ in L_{in} by u. In the outer loop, we load entries $(\underline{u_1} \to v, d), (\underline{u_1} \to v', d), \ldots, (\underline{u_2} \to v, d)$, in L_{out} (sorted by u_i) and entries $(u_1 \to \underline{v}, d), (u_1 \to \underline{v'}, d), \ldots, (u_2 \to \underline{v}, d), \ldots$, in L_{in} (sorted by u_i) into Buf_{all}. In the inner loop, we load entries L_{prev} (sorted by u) into Buf_{prev} in batch. For entries in Buf_{all} and Buf_{prev}, we try to apply the first generation rule to generate candidate out-entries. For generating in-entries in L_{cand}, the algorithm is similar. We summarize the I/O cost for each iteration, including the candidate generation process and the pruning process.

Lemma 7. *In each iteration, let L be the set of entries generated and survived from pruning from all previous iterations. Let L_{prev} be the set of entries generated and survived from pruning from the previous iteration. Let L_{cand} be the set of entries generated in the current iteration. The total I/O cost for candidate generation and pruning is*

$$O(\lceil (|L| + |L_{cand}|)/M \rceil \cdot scan(|L| + |L_{cand}|)). \quad (2)$$

Proof: We first consider the I/O cost for generating out-entries in L_{cand}. Before loading entries into memory, we sort in-entries in L, which needs $sort(|L|)$ I/Os. In the nested loop, all the entries in L_{prev} are loaded into buffer $\lceil |L|/(M-2B) \rceil$ times, and all entries in L_{all} are loaded into buffer once. The time for loading entries is thus $O(\lceil |L|/M \rceil \cdot scan(|L_{\text{prev}}|) + scan(|L|))$. We then consider the cost for merging and pruning out-entries, the I/O cost is $O(\lceil (|L|+|L_{\text{cand}}|)/M \rceil \cdot scan(|L|+|L_{\text{cand}}|))$ from Lemma 4. Note that because $\log_{M/B} |L|/B \leq |L|/B$ when $|L|$ is sufficiently large, we have $sort(|L|) = O(|L|/B \cdot \lceil |L|/M \rceil)$. Thus, the total I/O cost for generating, pruning and merging out-entries is $O(\lceil (|L| + |L_{\text{cand}}|)/M \rceil \cdot scan(|L| + |L_{\text{cand}}|))$. The analysis for generating, pruning and merging in-entries is similar. In conclusion, the total I/O cost in one iteration is $O(\lceil (|L| + |L_{\text{cand}}|)/M \rceil \cdot scan(|L| + |L_{\text{cand}}|))$.

B. Discussion

Practical issues: To avoid generating too many candidate entries in one iteration, Jiang et al. [7] suggest using entries in L with distance equals to 1 to construct candidate entries. We adopt their suggestion. In the first ten iterations of the *batch update* method, while applying generation rules, we ignore entries in L with distance larger than 1.

Utilizing L^{t-1}. By inserting entries in L^{t-1} into L in the initialization process, the *batch update* method may generate entries with distance greater than 2^i in the i-th iteration, which is impossible in the *Hop-Doubling Labeling* algorithm. However, utilizing L^{t-1} also introduces some extra cost. In every iteration, the size of L in the *batch update* method is no smaller than that in the *Hop-Doubling Labeling* algorithm.

V. Distance Queries on the Most Update Network

Here, we show how to answer distance queries toward the most updated graph G with label L^{t-1} for graph G_{t-1}. This is useful when we are running the *batch update* method, or when we decide not to update labels stored on disk.

Let $E_{\text{new}} = E_t \backslash E_{t-1}$ be the set of new edges. Let $V_{\text{new}} = \{u | \exists e_{uv} \in E_{\text{new}}, \text{or } \exists e_{vu} \in E_{\text{new}}\}$ be the set of endpoints of new edges. To answer the query about $d(s,t)$, we construct a weighted *query graph* $G_{\text{Q}} = (V_Q, E_Q)$ such that $d_{G_Q}(s,t) = d(s,t)$. The *query graph* contains two types of edges.

- *Update-related*: Edges in E_{new} are in G_{Q}. Moreover, for $v \in V_{\text{new}}$, every entry in $L_{\text{out}}^{t-1}(v)$ and $L_{\text{in}}^{t-1}(v)$ corresponds to an edge in G_Q. For example, if entry $(u,d) \in L_{\text{in}}^{t-1}(v)$, there is an edge from u to v with distance d in G_Q.
- *Query-related*: For node $v \in V_{\text{new}}$, there is an edge from s to v with distance $\texttt{QUERY}(L^{t-1}, s, v)$ and an edge from v to t with distance $\texttt{QUERY}(L^{t-1}, v, t)$. Moreover, G_Q contains an edge from s to t with distance $\texttt{QUERY}(L^{t-1}, s, t)$.

Using the fact that L^{t-1} is a 2-hop labeling of G_{t-1}, it is easy to verify that, with all *update-related* edges, we have $d_{G_Q}(u,v) \geq d(u,v)$ for all $u, v \in V_{\text{new}}$. *Query-related* edges are *"shortcuts"* from s to every node in V_{new}, *"shortcuts"* from every node in V_{new} to t, and the *"shortcut"* from s to t. Distances of these *"shortcuts"* are distances in G_{t-1}. The existence of *query-related* edges further ensure that $d_{G_Q}(s,t) = d_G(s,t)$. Formally, we have the following theorem claiming the *correctness* of the *query graph*.

Theorem 3. *The distance from s to t in G_Q is $d(s,t)$.*

Algorithm. When there is a new query about $d(s,t)$, the construction of G_Q could be fast. Note that the *update-related* edges are not related to any particular query. Thus, we could assume that we have loaded all *update-related* edges into memory beforehand. When there is a new query, we load $L_{\text{out}}^{t-1}(s)$ and $L_{\text{in}}^{t-1}(t)$ from disk. With $L_{\text{out}}^{t-1}(s)$, $L_{\text{in}}^{t-1}(t)$, and $L^{t-1}(v)$ for all $v \in V_{\text{new}}$, we could construct *query-related* edges in memory. Finally, we run the bidirectional Dijkstra algorithm on the query graph to get $d(s,t)$.

Theorem 4. *The I/O cost for answering the above distance query is $O(1)$. Let h be the average size of label (in and out-label) for a node in L^{t-1}, the CPU cost for answering a query is $O(|V_{new}| \log |V_{new}| + |V_{new}| h + |E_{new}|)$.*

Proof: To answer the query about $d(s,t)$, only loading $L_{\text{out}}^{t-1}(s)$ and $L_{\text{in}}^{t-1}(t)$ from the disk perform I/Os. Therefore, the I/O cost is $O(1)$. To construct the query graph, we need to answer $2|V_{\text{new}}|$ distance queries and the CPU complexity is $O(|V_{\text{new}}| h)$. In the query graph, the number of *update-related* edges is $O(|V_{\text{new}}| h + |E_{\text{new}}|)$ and the number of *query-related* edge is $O(|V_{\text{new}}| h)$. Thus, we have $|V_Q| = O(|V_{\text{new}}|)$ and $|E_Q| = O(|V_{\text{new}}| h + |E_{\text{new}}|)$. The total CPU complexity for answering a query is $O(|V_{\text{new}}| h + (|V_Q| \log |V_Q| + |E_Q|)) = O(|V_{\text{new}}| \log |V_{\text{new}}| + |V_{\text{new}}| h + |E_{\text{new}}|)$. ∎

VI. Experimental Results

We perform experiments on real world networks. First, we show the efficiency of the *single edge update* method and the *batch update* method. In particular, we show the importance of refinements for our *single edge update* method. Second, we show the I/O efficiency of our query algorithm, which runs based on the outdated labeling and a set of new edges. We implement our proposed algorithms and the *Hop-Doubling Labeling* algorithm in C++. Experiments are conducted using 4GB memory (so to create I/O activities) on a Linux machine with Intel 3.20GHz CPU and 7200 RPM SATA hard disk.

Datasets. Table I summarizes our datasets. We treat each dataset as a directed graph and we remove duplicated edges. Dataset *Enron-email* and *Youtube* have timestamp on edges indicating their arrival time. For these two datasets, we sort edges in the ascending order of their arrival time. For duplicated edges, we keep the one with the smallest timestamp. Then we let G_{t-1} contains the first half of the sorted edges and let other edges be new edges. For datasets without timestamp on edges, we treat the complete dataset as G_{t-1} and we obtain G_t by randomly inserting $100,000$ edges not in E_{t-1}.

TABLE I. REAL SCALE-FREE DATASETS

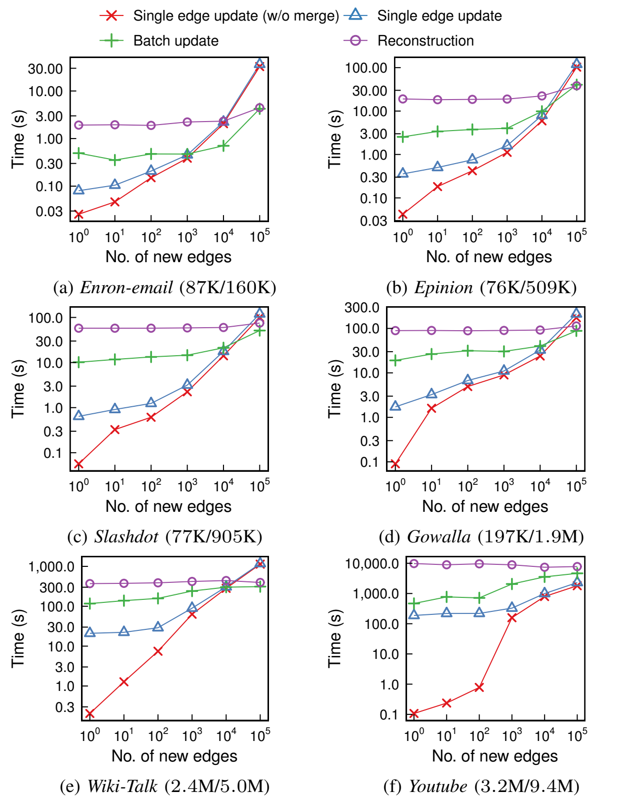
Dataset	$\lvert V\rvert$	$\lvert E_{t-1}\rvert$	timestamp	directed or not
Epinion [10]	76K	509K	no	directed
Slashdot [11]	77K	905K	no	directed
Enron-email [12]	87K	160K	yes	directed
Gowalla [13]	197K	1.9M	no	undirected
Wiki-Talk [14]	2.4M	5.0M	no	directed
Youtube [15]	3.2M	9.4M	yes	undirected

A. *Comparison among update methods.*

Disk-based update methods. We test the efficiency of our two update methods. For each dataset, we first construct the *canonical labeling* L^{t-1} using the *Hop-Doubling Labeling* algorithm and we record the rank r for later use. Then, we run the *single edge update* algorithm and the *batch update* algorithm. For the *single edge update* method, we adopt our *lazy patch merge* strategy and allocate 1% ($\approx$ 40MB) of the main memory as the *"prefecth buffer"*. We also run the *Hop-Doubling Labeling* algorithm to reconstruct a *canonical labeling* with re-computed ranks and let it be the benchmark.

Figure 3 shows the results on datasets with timestamps. Our experimental results suggest that, if the number of new edges is no greater than $100,000$, the *batch update* method is preferred over the reconstruction method because of the smaller label maintenance time. Moreover, when the number of new edges is relatively small, the *single edge update* method outperforms the other two methods. The cumulative patch generation time (i.e., *single edge update (w/o merge)*) grows almost linear with the number of new edges, which is consistent with our analysis of the patch generation phase. Moreover, the gaps between the *single edge update* time with and without the merge phase show that the *lazy patch merge* refinements significantly reduce the amortized *patch merge* cost over new edges.

Memory-based update methods. We compare the *Single edge update* method with the update method (*Dynamic PLL*) of the *Pruned-Landmarking Labeling* (PLL) [6]. Since both the *PLL* construction algorithm and *Dynamic PLL* are memory-based, we directly change our *single edge update* algorithm to a memory-based one for comparison. We did not use datasets



(a) *Enron-email* (87K/160K) (b) *Epinion* (76K/509K) (c) *Slashdot* (77K/905K) (d) *Gowalla* (197K/1.9M) (e) *Wiki-Talk* (2.4M/5.0M) (f) *Youtube* (3.2M/9.4M)

Fig. 3. Comparison among update methods and the reconstruction method.

Wiki-Talk and *Youtube* because they are too large for memory-based algorithms. For each dataset, we treat it as an undirected graph, and prepare 2-hop labeling of G_{t-1} for *Dynamic PLL* (resp. *single edge update* method) using the PLL algorithm (resp. *Hop-Doubling algorithm*). Then, we insert $100,000$ new edges in sequence using *single edge update* method and *Dynamic PLL*. Table II summarizes our experimental results. For four datasets tested, the update time of our *single edge update* method is comparable with that of *Dynamic PLL*. Suppose the 2-hop labeling before and after update is L_1 and L_2, the "label increase" is defined as $|L_2 - L_1|/|L_1|$. Table II shows that the average "label increase" of the *single edge update* algorithm is two to three orders of magnitude smaller than that for *Dynamic PLL*. This is mainly because if there is an outdated entry whose distance needs to be decreased, *Dynamic PLL* inserts a new entry instead of update the distance. Therefore, our experiments suggest that, to maintain a small label size, we should update outdated entries instead of insert new entries. The small "label increase" for the *single edge update* method also implies that we could keep label patches in memory for many edge insertions before we perform the *lazy patch merge*.

TABLE II. COMPARISON BETWEEN MEMORY-BASED UPDATE METHODS

Dataset	$\lvert V\rvert$	$\lvert E_{t-1}\rvert$	Single edge update		Dynamic PLL	
			Time (μs)	Label increase ($\cdot 10^{-6}$)	Time (μs)	Label increase ($\cdot 10^{-6}$)
Epinion	76K	509K	134	4.5	122	3201.7
Slashdot	77K	905K	215	3.0	251	2074.3
Enron-email	87K	160K	135	13.0	197	5846.7
Gowalla	197K	905K	290	5.0	201	3880.7

Refinements of *single edge update*. We show the performance gain of allocating a small buffer to prefetch labels of nodes with top ranks. Table III shows the speedup of the patch generation phase by using only 1% of memory as buffer. Note that in the extreme case where the buffer is large enough to load all entries, the disk-based label update algorithm essentially becomes memory-based. Therefore, the larger the buffer is, the higher speedup can be achieved. For a fixed buffer size, Table III shows that the patch generation phase tends to achieve higher speedup for smaller networks, which is consistent with our analysis. Moreover, even for the two largest datasets where the size of outdated labeling is larger than 1GB, applying the *label prefecth* refinements could still speedup the patch generation. In conclusion, we suggest adopting the *label prefetch* refinements in general. As to how to select the buffer size, it may depend on other factors such as whether there are other jobs sharing resources of the machine. If possible, one may allocate a buffer as large as possible so that the *single edge update* method achieves the highest speedup.

TABLE III. SPEEDUP OF PATCH GENERATION (BUFFER SIZE: 40.96MB)

Dataset	$\|L_{t-1}\|$	**Speedup verses number of new edges**			
		100	1000	10000	100000
Epinion	68MB	22.85	24.74	9.28	4.78
Slashdot	138MB	37.24	24.65	7.91	7.26
Enron-email	60MB	6.15	4.97	4.98	4.87
Gowalla	190MB	8.00	8.18	5.00	3.73
Wiki-Talk	1.4GB	1.39	1.22	1.19	1.20
Youtube	3.4GB	1.34	1.48	1.15	1.36

B. Distance Query on the most update network

For each dataset, we construct the *canonical labeling* for graph G_{t-1} using the *Hop-Doubling Labeling* algorithm. Then, we test our query algorithm by measuring the average query time when there is different number of new edges. For each experiment, we answer $5,000$ randomly generated distance queries and show the average query time. In our experiments, we clear the filesystem memory cache before answering each query. By doing so, we are actually measuring the worst case query time because every I/O request results in a physical I/O.

Figure 4 shows the performance of our query algorithm. When the number of new edges is small, say below 1% of the previous number of edges, the number of new edges has a negligible impact on the query time. The reason is that the CPU time is small as compared with the I/O cost for loading labels for two nodes. Note that for all disk-based 2-hop labeling, answering query about $d(u, v)$ requires loading $L_{\text{out}}(u)$ and $L_{\text{in}}(v)$ from disk. Therefore, our query performance is comparable with the query performance where the 2-hop labeling is up-to-date. For the four smaller datasets, when the number edges exceeds 10^4, the query time increases with the number of new edges because the CPU cost starts to factor in. But the overall query time is still acceptable. In general, our experimental results demonstrate that our proposed query algorithm is a good approach to handle distance query while the system is also processing the label update algorithms.

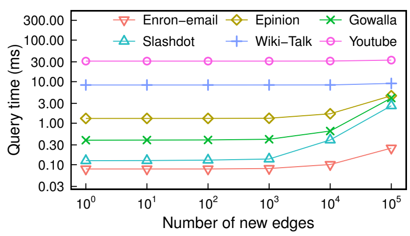


Fig. 4. Results of query algorithm on real datasets.

VII. Conclusion

In this paper, we address the problem of efficiently answering exact distance queries on *disk-resident scale-free "dynamic" graphs*. The query processing is based on the *canonical labeling*, which is a special case of the 2-hop labeling. To answer exact distance queries efficiently on dynamic graphs, we present two methods to *"incrementally update"* the general *canonical labeling*. The two update methods, namely the *single edge update* method and the *batch update* method, have significantly different designs and each has its own merits. For the scenario where one has decided not to update the disk-based labeling, we propose a query algorithm that returns shortest distance toward the latest network based on out-dated labeling and a set of new edges, which is also a good approach to handle exact distance queries while the system is updating the labeling. We conduct extensive experiments on real scale-free networks to test our proposed methods. Experimental results demonstrate that it is possible to update *disk-based canonical labeling* efficiently and incrementally, instead of reconstructing the labeling from the scratch. Moreover, we show via experiments that one could efficiently answer distance queries even with out-dated disk-based *canonical labeling*, given that the number of new edges is not too large. Considering that many large-scale networks cannot fit into memory of a typical PC, we believe that studying how to incrementally maintain 2-hop labeling using external memory algorithms or distributed algorithms is a challenging yet promising future direction.

Acknowledgment: This work is supported in part by the RGC Grant 415013.

REFERENCES

[1] C. Sommer, "Shortest-path queries in static networks," *ACM Computing Surveys*, vol. 46, no. 4, 2014.
[2] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. on Computing, 32(5)*, 2003.
[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," in *Algorithms–ESA 2012*, 2012.
[4] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong, "Is-label: an independent-set based labeling scheme for point-to-point distance querying," *Proc. of the VLDB Endowment*, vol. 6, no. 6, 2013.
[5] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *Proc. of SIGMOD'13*, 2013.
[6] ——, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *Proc. of WWW'14*.
[7] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu, "Hop doubling label indexing for point-to-point distance querying on scale-free networks," *Proc. of the VLDB Endowment*, vol. 7, no. 12, 2014.
[8] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *CACM*, vol. 31, no. 9, 1988.
[9] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, 1999.
[10] M. Richardson, R. Agrawal, and P. Domingos, "Trust management for the semantic web," in *Proc. of ISWC'03*. Springer, 2003.
[11] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, 2009.
[12] B. Klimt and Y. Yang, "The Enron corpus: A new dataset for email classification research," in *Proc. ECML'04*, 2004.
[13] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and mobility: user movement in location-based social networks," in *Proc. of KDD'11*.
[14] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Signed networks in social media," in *Proc. of the SIGCHI'10*, 2010.
[15] A. Mislove, "Online social networks: Measurement, analysis, and applications to distributed information systems," Ph.D. dissertation, Rice University, 2009.