Pay-as-you-go Maintenance of Precomputed Nearest Neighbors in Large Graphs

Tom Crecelius
Max Planck Institute for Informatics
Saarbrücken, Germany
tcrecel@mpi-inf.mpg.de

Ralf Schenkel
Saarland University
Saarbrücken, Germany
schenkel@mmci.uni-saarland.de

ABSTRACT

An important building block of many graph applications such as searching in social networks, keyword search in graphs, and retrieval of linked documents is retrieving the transitive neighbors of a node in ascending order of their distances. Since large graphs cannot be kept in memory and graph traversals at query time would be prohibitively expensive, the list of neighbors for each node is usually precomputed and stored in a compact form. While the problem of precomputing all-pairs shortest distances has been well studied for decades, efficiently maintaining this information when the graph changes is not as well understood. This paper presents an algorithm for maintaining nearest neighbor lists in weighted graphs under node insertions and decreasing edge weights. It considers the important case where queries are a lot more frequent than updates, and presents two approaches for transparently performing necessary index updates while executing queries. Extensive experiments with large graphs, including a subset of Twitter's user graph, demonstrate that the overhead for this maintenance is small.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and networks; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

Keywords

shortest paths, incremental APSD, databases

1. INTRODUCTION

1.1 Motivation

An increasing amount of applications need to manage large graphs with millions of nodes and billions of edges. Examples include user graphs in social networks such as Facebook, Myspace, or Twitter, knowledge graphs in large knowledge bases such as DBPedia or YAGO, or keyword search in large databases. A common problem that many applications need to solve is accessing transitive neighbors of a specific node ordered by increasing distance, which is a key building block of algorithms for aggregated search in social networks [4, 24], explorative search of connections in knowledge bases [17], or retrieval in linked documents [13].

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

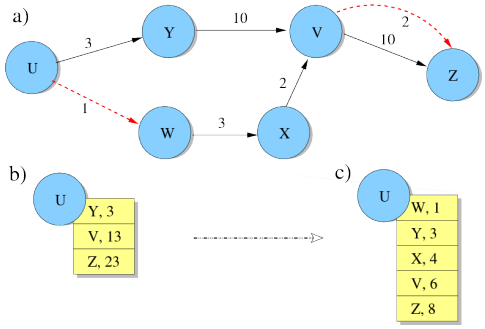


Figure 1: Example graph: A single inserted edge at U changes its entire nearest neighbor list and a decreased edge weight at V affects almost the entire graph because of changes in shortest paths for all predecessor nodes of Z.

For small graphs that fit into main memory, incrementally retrieving neighbors of a node can be efficiently implemented by running Dijkstra's algorithm [10]. However, this is not a valid option for large, disk-resident graphs as each access to a node corresponds to one disk access. For static graphs that do not change, a more efficient solution is to precompute, for each node, the distance to all other nodes (known as all-pairs shortest distance problem), and store this list of nodes in increasing order of distance. While precomputing this information can be expensive for large graphs, it is an offline, one-time operation. Accessing all neighbors of a node now requires only a few disk accesses to load this list.

This solution, however, turns prohibitively expensive when the graph changes over time. When a new edge from a node u is inserted or the weight of an existing edge starting at a node v is decreased, i.e., the distance to the edge's destination node is reduced, the precomputed nearest neighbor list of u or v, respectively, needs to be recomputed: new nodes may have appeared in that list or the distance of already connected nodes may have decreased. Figure 1a) depicts an example graph and 1b) the corresponding initial neighbor list for node u. The dashed lines indicate two graph updates: a new edge with weight 1 is inserted at node u and the weight of the edge starting at v is reduced from 10 to 2. As a consequence, nodes w and x are now connected with u through the new edge and appear as additional entries in u's neighbor list. Furthermore, the path to v and z through the new edge is now shorter than the previously existing shortest path (i.e. the distance to v and z changed from 13 and 23 to 6 and 8, respectively), and, hence, the corresponding entries in u's nearest neighbor list need to be updated as shown in Figure 1c). Moreover, the nearest neighbor lists of *all nodes preceding an updated node* v may have to be recomputed as

a new edge or changed weight may affect the distances from these nodes to any node reachable through that edge. In our example the distance to node z changes indeed for all other nodes in the graph due to the decreased weight of the edge starting at v.

This problem, known as dynamic all-pairs shortest paths (APSP), has been intensively studied in the literature for only inserting / decreasing or deleting / increasing of edge weights in graphs or both (see Section 2 for an overview). However, this existing work focuses on maintaining in-memory data structures for graphs that fit into main memory, and therefore cannot be applied for large, disk-resident graphs. Additionally, the proposed algorithms attempt to update all affected precomputed information as soon as a new edge is inserted. In most applications, this proactive update is not needed, as some lists may not be queried for a long time, and additional updates in the future may happen before such a query arrives.

This paper proposes an algorithm for incrementally maintaining precomputed nearest-neighbor lists under edge addition and decreasing edge weights that defers updates of the lists as long as possible, namely to the point where a query attempts to read an entry from such a list that is not up to date. Any query that does not touch updated edges will efficiently read the precomputed list of neighbors. We provide two alternative versions of the algorithm. The first version maintains neighbor lists by merging complete neighbor lists at query time. The second version incrementally merges lists while processing a query, reading a variable number of entries from each list. Our algorithm has been developed with friendship graphs of social networks in mind where the insertion of friendship connections is much more frequent than their deletion. Moreover, we are aiming at operations on graphs that introduce "better" paths, not operations that reduce the quality of existing paths. We evaluate our proposed algorithms with two excerpts of real social networks.

1.2 Notation

We consider a sequence $G^0 \dots G^T$ of *directed weighted graphs* with a constant set V of nodes and a varying set of edges E^t and edge weights. $G^t = (V, E^t)$ does not contain any self edges. For each edge $e = (v_i, v_j) \in E^t$, there is an edge weight $w^t(e) \geq 0$. We consider the special case where two consecutive graphs G^t and G^{t+1} in the sequence differ by applying exactly one *edge update* on a single node u. We define an edge update for a node u at time $t+1$ as an operation applied on u in G^t such that either

- the weight of an edge starting at u in G^t has been decreased in G^{t+1}, i.e., $w^t(e) > w^{t+1}(e)$ for exactly one edge $e = (u, v)$ with $u, v \in V$ and $w^t(e') = w^{t+1}(e')$ for all other edges $e' \neq e$, or
- a single edge not existing in G^t has been added to u in G^{t+1}, i.e., $\exists!\ e = (u, v) : E^{t+1} = E^t \cup e\ with\ e \notin E^t$, $G^t = (V, E^t)$, $G^{t+1} = (V, E^{t+1})$ and $u, v \in V$.

Note: The restriction to a constant set of nodes has been made only to ease discussion. The retrieval of nearest neighbors for newly added nodes without edges is trivial as there are no neighbors at all. Hence, those nodes are not important when moving from G^t to G^{t+1}. However, as soon as there is a new edge for such a node, the problem description complies to the one mentioned above.

The distance along a path is defined as the aggregation of the weight of all edges on the path. The choice of edge weights depends on the application. For social networks, a number of proposals for measuring relationship strengths exist [24, 27].

We compute the weight of a path $p = (v_1, \dots, v_n)$ in G^t as

$$w^t(p) = \sum_{i=1}^{n-1} w^t(v_i, v_{i+1})$$

and define the weight of an arbitrary pair of nodes u and v in G^t as

$$w^t(u, v) = \min_{p=(u=v_1,\dots,v_n=v)} w^t(p)$$

Note that other notions of path weights can be incorporated in this model as well, for example maximization or the product of edge weights (if edge weights are between 0 and 1), which would correspond to transitive relationship strengths in social networks.

1.3 Problem Statement

For a given node u, we want to access the descendants v of u in graph G^t in ascending order of $w^t(u, v)$. We denote this operation as $query^t(u)$, where t corresponds to the the most up-to-date time with respect to the occurrence of updates. Hence, we are always interested in the most up-to-date nearest neighbors of a node, and no historical queries are possible or allowed. A query that retrieves only the first k descendants of u is denoted as $query^t(u, k)$.

For a node u, let denote $t(u)$ the time where all edge updates that occurred at a time $t' \leq t(u)$ have already been applied on u, such that u is up-to-date with respect to $G^{t(u)}$. We assume that we have already precomputed nearest neighbor lists for each node u, represented as lists $L(u)$ of u's descendants v in the graph $G^{t(u)}$ and sorted in ascending order of $w^{t(u)}(u, v)$.[1] Hence, each list for a node u is correct with respect to the node's specific time $t(u)$. We further assume that we know, for each node u, the edge updates that have been occurring for u since time $t(u)$, i.e. u's newly added or modified descendants since $t(u)$.

This paper aims at answering queries of the form $query^t(u)$ at time $t \geq t(u)$ by sequentially reading available precomputed information and, at the same time, updating this precomputed information on the fly when necessary.

Example. Assume that Figure 1a) depicts the initial graph G^0 at time 0 when neglecting the edges visualized as dashed lines. The corresponding precomputed list for node u at time $t(u) = 0$ is shown in Figure 1b). At time 1, the new edge $e_1(u, w)$ with weight 1 is added and at time 2, the weight of the edge $e_2 = (v, z)$ decreases from 10 to 2. Moreover, for each other node in G, there's also a precomputed list (not shown in the figure) for time 0 (or any later time 1 or 2). A $query^t(u)$ with $t = 2$ aims at retrieving u's descendants at time 2 and, equally, at updating u's list $L(u)$ by considering updated information about directly connected edges to u and by propagating updated information from lists $L(v)$ of descendants v while they are sequentially discovered in $L(u)$. The resulting nearest neighbor list is shown in Figure 1c).

2. RELATED WORK

The majority of work on indexing graphs has focused on compactly storing reachability information. Here, the proposed approaches include 2-hop covers [6, 25], 3-hop covers[15, 16], and storing tree- and non-tree edges of a graph separately [26]. Only a few proposals exist for efficiently updating such an index [3, 25].

Efficiently representing distances in such a compact index is a difficult problem, and mainly two classes of solutions have been proposed: adding distance information to 2-hop covers [2, 5, 6, 25], and building a shortest-path index by storing multiple BFS trees [28]. Gubichev et al [14] use path sketches to compute approximate distances and shortest paths between two nodes. Only [25] considers the problem of incremental index maintenance.

The problem of maintaining all-pairs shortest path information under certain classes of graph updates (like increasing or decreas-

[1] Imagine our algorithm is used not before the memory footprint of the graph reaches a certain threshold or the global recomputation of the graph exceeds a certain amount of time.

ing edge weights) is also an active topic in the algorithms community [1, 9, 12, 21, 8, 18, 23]. Here, solutions usually focus on graphs that fit in main memory. King [18] proposes a fully dynamic algorithm for APSP in directed graphs with integer edge weights bounded by b, which requires constant time to determine the distance between two arbitrary nodes. It provides amortized cost of $O(n^2\sqrt{bn})$ for a series of updates in a graph with n nodes [19], and it can be modified to retrieve neighbors of a node in ascending order of distance. Frigoni et al. [12] propose an algorithm for fully dynamic APSP in directed graphs with real weights; here, the amortized complexity of an update is $O(m \cdot \log n)$ in a graph with n nodes and m edges. Demetrescu and Italiano [9] experimentally compare several algorithms for the dynamic all-pair shortest path problem, namely [8, 18, 23] on random and real-world graphs of up to 3,000 nodes. Unlike these proposals, our method assumes that the graph is too big to store in memory and needs to be kept on disk, so traversing the graph as these methods do is not viable here. To the best of our knowledge, Meyer's work [21] is the only that considers graphs stored on external memory; however, it does not precompute any information, but performs a BFS traversal of the graph to determine node distances.

The problem of incremental maintenance of precomputed transitive closures and distances in databases was considered in [11, 22]. They update the complete set of precomputed information after a new edge was inserted or an existing edge was deleted. It is incremental in the sense that it does not need to recompute from scratch, but can modify the existing information. In contrast, our proposed approach is also incremental, but considers only a subset of the precomputed information affected by an update, and defers maintenance to the point where queries access information that is potentially out-of-date.

3. ALGORITHMIC OVERVIEW

3.1 Data Structures

We will make use of the following data structures:

- **L(u)** For each node u, $L(u)$ is the list of u's descendants in graph $G^{t(u)}$ in ascending order of distance and is initially precomputed and stored on secondary storage. Each entry in list $L(u)$ is a pair (v, w), representing a shortest path from u to node v with weight w.
  We use the short-cut notation $L(u, v)$ to refer to the weight w of a path from u to v stored in $L(u)$. For notational simplicity, we assume that $L(u, v) = \infty$ when there is no path from u to v. Moreover, $L(u)$ provides the operations POS(u, v), denoting the position of v in $L(u)$, and REPLACE(v, w'), which replaces the entry (v, w) in $L(u)$ with (v, w') or adds it when there is no such entry.
  The operations POS(u, v), REPLACE(v, w) and $L(u, v)$ only consider prefixes of lists $L(u)$ which have already been traversed and, thus, are available in main memory. Hence, finding or replacing entries is cheap.
- **t(u)** For a node u, the in-memory timestamp $t(u)$ keeps the time when $L(u)$ was last updated.
- **tp(u)** The timestamp validity pointer $tp(u)$ is an offset into $L(u)$ and points to the position up to which all nearest neighbors are guaranteed to be correctly identified with respect to the timestamp $t(u)$.
- **OP(u)** The in-memory set $OP(u)$ keeps for each node u the set of edge updates for u that have not yet been applied on u since $t(u)$. Entries are triples (v, w', t) where v is the target node of the updated edge, w' the updated weight for the edge (u, v) and t the time when the update occurred. For multiple updates of the same edge since $t(u)$, only the latest update is stored in $OP(u)$. Whenever an entry is read from $OP(u)$, it is immediately removed and the corresponding main memory is freed.

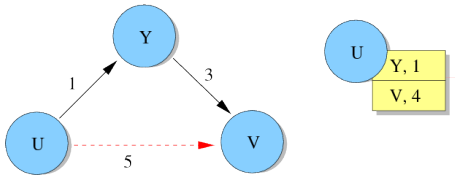


Figure 2: Example graph showing that a weight of an indirect path is lower than the one of a new edge update (the dashed line).

Initially, all $L(u)$ are precomputed for G^0, $t(u) = 0$, $tp(u)$ points to the end of $L(u)$, and $OP(u) = \emptyset$. The precomputed lists are stored on disk and loaded in batches when they are accessed. Therefore, REPLACE(v, w') requires a special handling: Assume that the in-memory prefix of $L(u)$ does not contain an entry for v and there is a new edge (u, v) which is applied during the query processing and inserted by REPLACE(v, w'). Since $L(u)$ is sorted in ascending order of distance, all entries already read from disk have a lower weight than the remaining ones on disk. However, this is not necessarily true for a new edge (u, v). The weight w' of such an edge might actually be higher than the one of an indirect path from u to v whose edges are summing up to a smaller value $w < w'$ and which is already stored in the part of $L(u)$ still on disk. An example for this scenario is given in Figure 2. The indirect path from u over y to v has a lower weight $w = 1 + 3$ than the weight $w' = 5$ of the new direct edge (u, v).

Further assume that only the first entry of $L(u)$ is already in memory. Hence, when REPLACE(v, w') is called for $L(u)$, the in-memory part of $L(u)$ does not contain v, but the disk-based part contains an entry with a lower weight than the one of the new edge. We therefore buffer the new entry for v to $L(u)$ in memory and merge it with entries loaded in the next batches from disk, keeping only the entries with smallest weight if a node appears more than once. We remember the last weight $max(u)$ that has been read from disk for $L(u)$, and load the next batch from disk when reading an in-memory entry with an higher weight than $max(u)$.

3.2 Base Algorithm

Our algorithm framework builds on the following principles. Edge updates are not included in the precomputed lists when they happen but delayed to the last possible point, which is when they have an effect on the result of a query. We then process them in an on-the-fly manner during processing the query. To achieve this, a query $query^t(u)$ reads the precomputed list $L(u)$ until it encounters an added or modified edge; in that case, just the necessary updates are incorporated into the precomputed data structures.

Our base algorithm NEIGHBORS(u, k) for processing a query $query^t(u, k)$, shown in Listing 1, uses two core operations, UPDATE and MERGE:

- UPDATE(u): This method incorporates all edges updates in $OP(u)$ into $L(u)$, but does not deal with transitive paths that may have changed due to the modified edges.
- MERGE(u, v): This method takes care of transitive paths from u to other nodes passing through v by connecting paths starting at v. In this way, updates located at even distant nodes propagate to u.

We will later introduce two different implementations for these two methods. The base algorithm successively determines the closest k

```
NEIGHBORS(u, k)
 1  L ← ∅ ; i ← 0
 2  if OP(u) ≠ ∅ then UPDATE(u)
 3  R(u).RESET
 4  while (v ← R(u).NEXT) ≠ NL
 5      do
 6          L[i] ← v ; i ← i + 1
 7          if i = k then Break
 8          if OP(v) ≠ ∅ then UPDATE(v)
 9          if NEEDS-MERGE(u, v) then MERGE(u, v)
10          if i > tp(u) then tp(u) ← i
11  return L

NEEDS-MERGE(u, v)
1  return t(u) < t(v) or (pos(u, v) ≥ tp(u) and t(v) > 0))
```

Listing 1: Base algorithm for retrieving nearest neighbors of a node

neighbors of u. It first updates u itself to include any modified edges in $L(u)$ and then sequentially traverses $L(u)$, accomplishing update and merge operations for each seen node if necessary. An update of a node v, i.e., including new or modified edges of v into $L(v)$ that it does not yet include, is needed if and only if $OP(v)$ is not empty. The algorithm uses an operator $R(u)$ to sequentially read entries from $L(u)$ with operations NEXT for reading the next entry from the list and RESET for resetting the operator to the beginning of the list. The implementation of this operator depends on the implementation of UPDATE and MERGE; we will explain it later when discussing the two implementations.

The function NEEDS-MERGE(u, v) (also shown in Listing 1) determines if there are paths from u through v that are shorter than those already included in $L(u)$. This decision is made based on the timestamps of u and v and the timestamp validity pointer $tp(u)$. A merge is necessary if one of the two following conditions holds:

1. Whenever $t(v)$ is greater than $t(u)$, v must have seen some more recent updates than u, so there could be shorter paths through v created by these updates.
2. As soon as we reach $tp(u)$ in $L(u)$, we do not know if subsequent entries in $L(u)$ are correct (checks for update and merge are applied only to entries prior to $tp(u)$), so every node v that we encounter with $t(v) > 0$ (i.e., that had at least one update since we built the index) may introduce shorter paths.

Before returning the k nearest neighbors, $tp(u)$ can be safely increased to k as those are guaranteed to be correct.

Note: Only for ease of presentation, NEIGHBORS(u, k) computes the best k neighbors of u before returning the results. We introduce the algorithm in this way for a clearer pseudo code. It is, however, simple to change it to an operator-based architecture where results can be polled one at a time, analogous to the usage of the $R(u)$ operator to read from $L(u)$.

We will now introduce two implementations of this framework, a static combination of lists and an incremental approach, that differ only in their implementation of UPDATE, MERGE, and the NEXT method of the operator $R(u)$.

4. EAGER PROPAGATION APPROACH

Our first approach, coined *Eager Propagation (EAP)*, eagerly updates $L(u)$ while traversing it by propagating all available information from modified lists of descendant nodes when they are encountered. It maintains the invariant that an edge update is either stored in $OP(u)$ or its effects are included in $L(u)$. We now explain how UPDATE and MERGE are implemented in *EAP*.

```
UPDATE(u)
1  maxTS ← max{t|(v, w′, t) ∈ OP(u)}
2  foreach (v, w′, t) ∈ OP(u)
3      do
4          if w′ < L(u, v)
5              then L(u).REPLACE(v, w′)
6                   MERGE(u, v)
7  t(u) ← maxTS ; tp(U) ← 0 ; OP(u) ← ∅
```

Listing 2: Implementation of UPDATE in *EAP*

```
MERGE(u, v)
1  foreach x ∈ L(v)
2      do
3          if L(u, v) + L(v, x) < L(u, x)
4              then L(u).REPLACE(x, L(u, v) + L(u, v))
5  t(u) ← max{t(u), t(v)} ; tp(u) ← pos(u, v) + 1
```

Listing 3: Implementation of MERGE in *EAP*

4.1 EAP.UPDATE

Listing 2 shows the UPDATE(u) method in *EAP*. It iterates over all edge updates in $OP(u)$. For each such edge update (v, w', t) at time t and weight w' of edge (u, v), it checks if v's current weight w in $L(u)$ is larger, and replaces the entry in this case with the new one. (This includes the case when v is not yet included in $L(u)$ since $L(u, v) = \infty$ then.) If we modify the entry for v, we call MERGE(u, v) to take care of transitive paths through v. Finally, we set the timestamp $t(u)$ of u to the highest timestamp of any of the edge updates, and reset the timestamp validity pointer $tp(u)$ to the beginning of the list.

4.2 EAP.MERGE

All nodes x in $L(v)$ are connected to u through v and their distance from u is at most $L(u, v)+L(v, x)$ if that path is also a shortest path. The purpose of MERGE(u, v) (shown in Listing 3) is (a) to identify all nodes x in $L(v)$ that are either not yet contained in $L(u)$ or whose current weight in $L(u)$ is bigger than $L(u, v) + L(v, x)$, and (b) to replace those nodes in $L(u)$ accordingly. When the two lists are merged, we increase the timestamp $t(u)$ of u to the highest timestamp of both nodes. We set u's timestamp validity pointer $tp(u)$ to the next entry following v in $L(u)$.

4.3 EAP.NEXT

Implementing the operator to read from $L(u)$ is very simple with *EAP* since (1) any edge update is integrated directly into $L(u)$, and (2) it can take place only after the last position read in $L(u)$. Therefore, the NEXT method of the operator $R(u)$ simply needs to keep track of the current reading position and increases it with each call, reading the entry at that position in $L(u)$. (Remember that we may have to first load further entries from disk depending on the weight of that entry before we can return it, see the discussion about REPLACE(v, w') in Section 3.)

4.4 Problems and Improvements of EAP

The *EAP* approach needs to fully load from disk all lists that are merged with $L(u)$, so it comes with big performance penalties when these lists are huge and loaded from a slow storage backend. This is the price we need to pay to ensure that we read the correct

entries no matter how far we read into $L(u)$, and this is also the case when we write the updated $L(u)$ back to disk after the query.

Unfortunately, this is true even if we usually read only a few neighbors in each query. However, if it is possible to limit the maximal number of neighbors that any query can read, we can considerably improve performance. Denoting this limit by $maxk$, MERGE(u, v) needs to consider only the first $maxk$ entries of the list since no query will ever access more entries. In fact, it is even enough to consider at most $maxk - pos(u, v)$ entries since the entries up to v in $L(u)$ will not be affected anyway. We could additionally stop merging when the resulting distances are not smaller than the $maxk$'s current entry of $L(u)$.

5. LAZY PROPAGATION APPROACH

The *EAP* approach from Section 4 and especially its implementation of the MERGE operation needs to access complete precomputed lists. As lists are often very long and queries access only small prefixes of them, it often causes way more work than necessary. We now present the *Lazy Propagation (LAP)* approach that merges lists on the fly as entries are read, improving performance especially for queries that read only a small number of results. The main differences to *EAP* are as follows:

- UPDATE(u) considers all edge updates from $OP(u)$ but does not immediately modify $L(u)$. Instead, the target nodes of updated edges are added to a priority queue $PQ(u)$ with the edges' weights as keys. The queue keeps possible candidates for the next node to return.
- MERGE(u, v) considers transitive paths from u through v. In contrast to *EAP*, $L(v)$ is not completely merged into $L(u)$ but only its first entry x is added to the priority queue $PQ(u)$ together with its weight that is computed as $L(u, v)+L(v, x)$.

The invariant that the *LAP* approach maintains is the following: For all nodes u, any edge that was modified or added since $L(u)$ was created is either stored in $OP(u)$, has an entry in $PQ(u)$, or its effects are fully integrated into the current version of $L(u)$.

LAP uses the following data structure in addition to those mentioned in Section 3.1:

- **PQ(u)** The priority queue $PQ(u)$ is actually a mergeable min-heap[7] that stores entries of the form (v, x) where v is a node and x is either v or a node from $L(v)$. An entry is uniquely identified and can be retrieved by its first component, in this case v. The key of such an entry denotes the weight w of the shortest path from u to x through v. Among the standard operations of such a data structure, we make use of ADD(v, x, w) to add entry (v, x) with key w, and EXTRACT-MIN to extract the current entry with minimal key. We extend the data structure by operations CONTAINS(v) to test if it contains an entry for v, and by REMOVE(v) to remove the entry for v.

Whenever the list $L(u)$ is read, the corresponding priority queue $PQ(u)$ is read from disk, too; correspondingly, after flushing an updated $L(u)$ back to disk, the current content of $PQ(u)$ is flushed to disk as well.

We will now explain UPDATE and MERGE for the *LAP* approach.

5.1 LAP.UPDATE

The implementation of UPDATE(u) in *LAP* is shown in Listing 4. This method considers all edge updates in $OP(u)$. For each edge update (v, w', t) in $OP(u)$, it adds an entry (v, v) with key w' to $PQ(u)$, removing any existing entry for v, if the weight of the new or updated edge (u, v) yields a lower path weight than any existing

```
UPDATE(u)
1  maxTS ← max{t|(v, w', t) ∈ OP(u)}
2  foreach (v, w', t) ∈ OP(u)
3       do
4          if w'(v) < L(u, v)
5            then if PQ(u).CONTAINS(v)
6                    then PQ(u).REMOVE(v)
7                 PQ(u).ADD(v, v, w')
8  t(u) ← maxTS ; tp(U) ← 0 ; OP(u) ← ∅

MERGE(u, v)
1  R(v).RESET
2  x ← R(v).NEXT
3  if PQ(u).CONTAINS(v)
4     then PQ(u).REMOVE(v)
5  PQ(u).ADD(v, x, L(u, v) + L(v, x))
6  t(u) ← max{t(u), t(v)} ; tp(u) ← pos(u, v) + 1
```

Listing 4: Implementation of UPDATE and MERGE in *LAP*

path from u to v. As depicted in Figure 2, an indirect path can have a lower weight than a new, direct edge.

The entry in $PQ(u)$ represents all shortest paths from u over v to nodes which are not yet known, hence, the paths currently end at v; we do not need to consider $L(v)$ at this point since we first need to deal with v itself. Next, we set the timestamp $t(u)$ of u to the newest timestamp of all new or modified edges for u, and set the timestamp validity pointer to the beginning of the list.

5.2 LAP.MERGE

MERGE(u, v) considers transitive paths from u to descendants of v taken from $L(v)$. In *LAP*, we do not directly merge the two lists, but use the priority queue. We first reset $R(v)$, the iterator on $L(v)$, and read the list's first node x, the closest neighbor of v. We now add the entry $(v, x, L(u, v)+L(v, x))$ to $PQ(u)$, representing the fact that x is the closest node that can be reached from u through any path through v with a weight $L(u, v)+L(v, x)$. As in *EAP*, we set the timestamp $t(u)$ of u to the maximum of the timestamps of u and v, and move the timestamp validity pointer $tp(u)$ right after the position of v in $L(u)$ since the correct node at this position is either already in place or exists in $PQ(u)$.

5.3 LAP.NEXT

Unlike the *EAP* approach where UPDATE(u) and MERGE(u, v) were expensive and NEXT was cheap, the majority of the work is now done in the NEXT operation (shown in Listing 5). As the timestamp validity pointer indicates the validity of entries in $L(u)$, we can immediately return the current entry from $L(u)$ as long as the read position is below $tp(u)$. The framework makes sure that we never read beyond $tp(u)$ without first updating $L(u)$, so this cannot happen. The only nontrivial case is when we are reading exactly at the position where $tp(u)$ points to. In this case, we have to identify the best candidate in $PQ(u)$ that is not yet part of the already read segment of $L(u)$, and read either this or the current entry in $L(u)$ at position $tp(u)$, depending on their weights. Identifying the best node from all candidates in $PQ(u)$ requires the following tasks:

1. Remove from $PQ(u)$ the current head triple (v, x, w) with the top candidate node x with weight w for a path from u to x over v.
2. If w is not smaller than the weight at the current reading posi-

```
R(u).NEXT
 1  ▷ pos: current reading position
 2  pos ← pos +1
 3  if (pos) = tp(u)
 4     then (v, x, w) ← PQ(u).EXTRACT-MIN
 5          while pos(u, x) < tp(u) or x = u
 6             do QUEUENEXT(u, v, x)
 7                (v, x, w) ← PQ(u).EXTRACT-MIN
 8          if (v, x, w) = NL
 9            then return L(u)[pos]
10          (y, z) ← L(u)[pos]
11          if L(u)[pos] = NL or w > z
12            then L(u).REPLACE(x, w)
13                 QUEUENEXT(u, v, x)
14            else PQ(u).add(v, x, w)
15          tp(u) ← tp(u) + 1
16  return L(u)[pos]

QUEUENEXT(u, v, x)
1  if NEEDS-UPDATE(x)
2     then UPDATE(x)
3  if NEEDS-MERGE(v, x)
4     then MERGE(v, x)
5  x ← R(v).NEXT
6  PQ(u).ADD(v, x, L(u, v) + L(v, x))
```

Listing 5: Implementation of NEXT in *LAP*

tion of $L(u)$, put it back to $PQ(u)$ and return the node from $L(u)$.

3. Otherwise, identify the next neighbor of v by reading from $L(v)$ and put it into $PQ(u)$; note that this may require updating and merging that node to reflect any updates not yet in $L(v)$.
4. Next, check if the current candidate x has already be seen in $L(u)$ at an earlier position with a lower weight. In that case, it is ignored, and we continue with step 1.
5. Check if x is equal to u. If so, we found a path with a cycle back to u, which we ignore by continuing with step 1.
6. Finally, we return x as this is the correct next neighbor of u.

5.4 Resolving Cycles

5.4.1 Why cycles cause problems

In *EAP*, we merge complete lists or list prefixes of fixed size into the list $L(u)$ of a queried node u. Cycles in the graph could therefore never cause problems. In *LAP*, however, we are considering several lists at the same time, trying to bring them up to date through UPDATE and MERGE operations. We illustrate the problem now with a very simple example with just two nodes.

Assume there are only two nodes a and b that do not have any edges in G^0, so both $L(a)$ and $L(b)$ are empty. Now we add edges $a \to b$ and $b \to a$ with weights $w_b = 1$ and $w_a = 3$, so a and b are mutual closest neighbors. Next, we query node a to retrieve $L(a)$. Our algorithm first updates a with the new edge to b by putting $(b, b, 1)$ into $PQ(a)$ (in UPDATE(a)), which will be at the head of $PQ(a)$ since it is the only entry. Thus, when our algorithm identifies the first neighbor of a by calling $R(a)$.NEXT, it will find b as top candidate in $PQ(a)$ and retrieves $(b, 1)$ as a result. Furthermore, it creates an entry in $PQ(a)$ for b containing b's nearest neighbor. Since a is b's nearest neighbor, this results in an entry $(b, a, 1 + 3)$ in $PQ(a)$. However, to identify that a is b's nearest neighbor, the same procedure as just described for a is executed for b, too. Hence, $PQ(b)$ finally contains the single entry $(a, b, 3 + 1)$. The scenario is depicted in Figure 3.

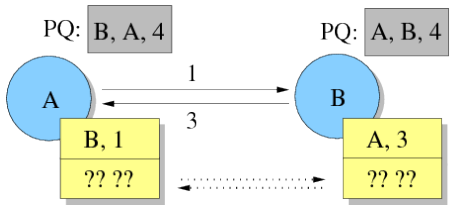


Figure 3: Example: cyclic graph. The next entry in PQ of each node cannot be processed because of the cycle.

The problem with cycles becomes obvious when trying to find the next neighbor of a. Then, the entry at the head of $PQ(a)$ is fetched, which is $(b, a, 4)$. As a is obviously not a valid neighbor, our algorithm tries to queue b's next neighbor. However, when trying to find b's next neighbor, we end up in the same situation as with a: the candidate for b's next neighbor is a's next neighbor. Hence, we are stuck in a cycle. Once such a cycle is identified, it is simple to break: as there are no other candidates left, each list ends at the current entry and there are no more next neighbors of a and b. The situation can get more complex when more nodes and longer lists are involved. We will now first describe how to detect cycles, and then how to break them.

5.4.2 Detecting Cycles

Identifying a cycle is rather simple. For a given query, we remember all nodes u for which the method QUEUENEXT(u, v, x, y) (see Listing 5) is called in a global data structure, e.g. a HashSet. When the method returns, we remove u from the data structure again. If u calls that method to queue a next entry from some other node's list and that node is already included in the data structure, we have identified a cycle.

5.4.3 Breaking Cycles

We mainly need to take care when extracting the current best entry from $PQ(u)$. If (v, x, w) is extracted from $PQ(u)$ and x belongs to a cycle, then we remove the entry and proceed to the next entry in $L(v)$ by calling QUEUENEXT(u, v, x), inserting it in $PQ(u)$ afterwards. We repeat this step until a candidate is found in $PQ(u)$ that does not belong to a cycle, or $PQ(u)$ is empty. When there is no more entry in $L(v)$, it is discarded from $PQ(u)$ and we repeat until a valid entry is found or $PQ(u)$ is empty.

5.5 Extension of LAP

Whenever $L(u)$ is loaded from disk for querying it, the corresponding priority queue $PQ(u)$ needs to be loaded and initialized as well. Especially when one of the previous queries read many entries of $L(u)$, this queue can be large, and it would be very expensive to open the lists for all nodes that have entries in $PQ)(u)$. Luckily, this is not necessary at the beginning but only when the queue entry for a node is pulled from the queue for the first time. To see why this is correct, assume that $PQ(u)$ contains an entry (v, x, w) when loaded from disk. This means that (1) v is contained in $L(u)$ since it is a descendant of u, (2) x is a descendant of v, (3) $L(u, v) \leq w$, and (4) all nodes in $L(v)$ up to x are already contained in $L(u)$. If any node in $L(v)$ up to x has seen a change since $PQ(u)$ was written to disk, this will be detected while traversing

$L(u)$, and necessary UPDATE and MERGE operations will be issued. When the entry for v is polled from $PQ(u)$, $L(v)$ is opened and read, ignoring any entries already seen. If the weight for x has decreased since $PQ(u)$ was written, it has already been seen in previous steps and will be ignored. Therefore, the serialized version of $PQ(u)$ on disk does not need to include the current candidate from each list, but simply entries of the form (v, w), where v is the note from whose list $L(v)$ entries are merged, and w is the last weight read from that list. Here, w serves as a threshold for opening the actual list $L(v)$ when running subsequent queries on $L(u)$.

6. EXPERIMENTAL EVALUATION

6.1 Setup

We evaluate our algorithm on excerpts of the following two social networks:

- **LT**: A combined corpus retrieved by two subsequent partial crawls of the user graph of the social book catalog LibraryThing, provided by the authors of [24]. It includes $7,793$ connected users and $28,853$ friendship edges (derived from explicit friends and users marked as having interesting libraries). Edge weights are defined as the Dice coefficient of the two users' tag sets.
- **Twitter**: A complete crawl of the user graph of the microblogging service Twitter which was used in [20]. It includes approximately 41 million users and 1.4 billion edges. As we did not have access to the tweets of the users, edge weights were assigned randomly from a uniform distribution.

Nodes in each graph are represented by integers. To form the initial graph G^0, we randomly removed edges according to the following pattern: For nodes with more than two edges, we removed 30% of the edges; for nodes with two edges, we removed one of them, and for nodes with only a single edge, we removed that edge for 50% of the nodes. We built the initial lists $L(u)$ for each node u using Dijkstra's algorithm and stored them in an Oracle database with appropriate indexes. The database runs on a Windows 2003 server with two dual-core AMD Opteron 2218 processors at 2.4 GHz and 32GB of main memory and a large network-attached disk array. All experiments, implemented in Java, were done on a Linux desktop PC with 4GB main memory and a 4-core Intel i5 650 CPU at 3.2GHz. Since the runtime costs are dominated by I/O accesses, the speed of the CPU is not crucial: in our single threaded experiments, a single core was barely used. For the bigger data set of Twitter, the total memory consumption of our algorithm reaches only 2.1GB after processing over 2 million queries.

Our algorithm is evaluated in the following way: We mix queries with updates at a fixed ratio (denoted as *upd-ratio* in the charts), i.e., after a fixed number of queries, one of the initially removed edges is added to the graph. A query retrieves the *top-k* nearest neighbors of a randomly chosen node. The order for adding edges was selected randomly once and is the same for all experiments.

For the evaluation, we measured the number of performed MERGE operations (#MRG), opened lists (#OL), the total number of list entries retrieved by sequential accesses (#SA) to the database, and the wall clock runtimes (RT) for each query. We report the results in charts where the x-axis denotes the number of the queries and the y-axis denotes the performance result. Additionally, the charts contain the moving average over intervals of 1000 queries, while only considering queries in the interval that indeed achieved a result set of *top-k*=200. Thus, quickly answered queries for nodes with no or only a few neighbors are neglected for the average computation.

6.2 Results for EAP on LT

Figure 4a shows the wall-clock runtimes (RT) and Figure 4b the number of sequential accesses (#SA) with the *EAP* approach. In total 63,572 queries were executed and every 100 queries one update was applied to the graph. The runtimes are not affected too much from #MRG and #OL (for *EAP*, #MRG=#OL) but mostly from #SA. In worst case, when each of the *top-k* friends is found in different lists by different merge operations, at most 200 lists are opened with *top-k*=200. Indeed, from the results in Figure 4c it is evident that EAP opens around 180 lists per query on average. This shows that the graph is tightly connected and a single update already influences many shortest path distances.

Furthermore, from Figures 4a and 4b it is evident that the runtimes are I/O-bound and increase rapidly with the number of read list entries. It takes over $20s$ to sequentially read around $60k$ entries from the involved friendship lists. The number of sequential reads is so large since the lists of transitive neighbors involved in a merge operation are long and, by the characteristics of *EAP*, have to be completely read during a MERGE operation.

Restricting Merge Operations to max-k.

Figures 4d–4i depict the results when merge operations are restricted to *max-k* entries. In each of the experiments *max-k* is set equal to a constant *top-k* value of 200 while the update ratio varies between 1 update per 1 and 100 queries. Depending on the update ratio, $17,438$ or $218,910$ queries were executed.

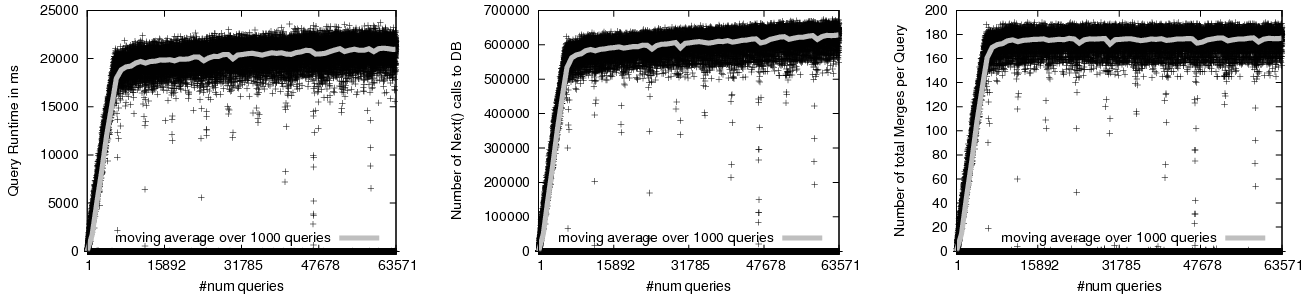
It can be observed that the different update ratios have only little effect on RT and #SA. For a query with an update ratio $1/1$, the maximum values for RT and #SA is $693ms$ or 2925 sequential accesses to the DB, respectively, and with an update ratio of $1/100$, the maximum value for RT is $723ms$ and 2130 for #SA. RT being slightly higher in the latter case although #SA being lower was most likely caused by network latency issues during the experiments. The average RT and #SA is fairly similar for both update ratios, too. The same is true for an update ratio of $1/10$, therefore, the charts are omitted. The drop in RT and #SA in Figures 4d and 4g after $8,456$ happens because all available edge updates have been applied to the graph at that time and, finally, the performance further improves when there are no more updates for a large number of queries. These experiments demonstrate that the modified *EAP* approach can handle different update loads and benefits quickly from long periods without updates.

Additionally, we evaluated the performance for an increased value of *max-k*=500 and an unchanged value of *top-k*=200 with 1 update per 100 queries (Figures 4f and 4i). As expected, RT and #SA increase because a larger prefix of the lists needs to be merged. However, the average RT/#SA keeps fairly stable at around $850ms$ and 2800 sequential accesses to the DB.

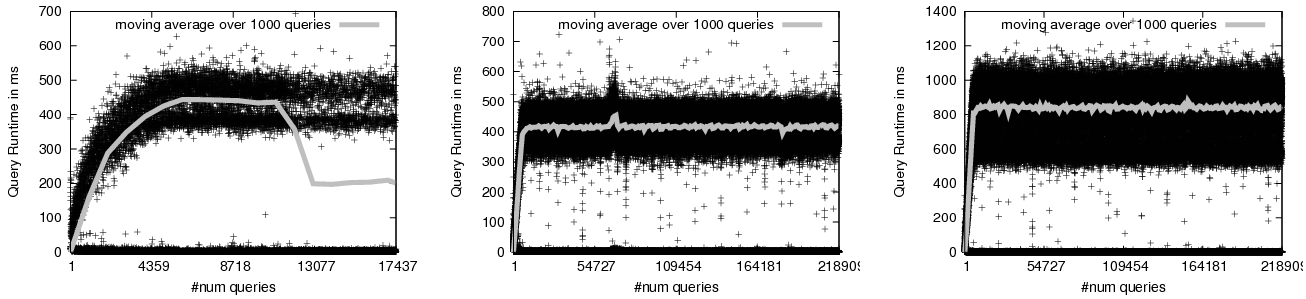
Even though one might expect for *max-k*=200 that there have to be 200 times 200 #SA for *max-k* opened lists and merge operations, this is usually not the case. Once a prefix of a list has been retrieved from the DB, it is kept in main memory until the end of the query and is reused whenever involved in another MERGE operation. For this reason, the relatively low #SA values also show that the graph is tightly connected. For tightly connected graphs, even a single update affects many shortest path distances. However, as shown, the amount of expensive I/O for retrieving the *top-k* nearest neighbors is still low with our algorithm. #SA could be further reduced and, thus, RT could be further improved when lists are cached for several queries.

6.3 Results for LAP

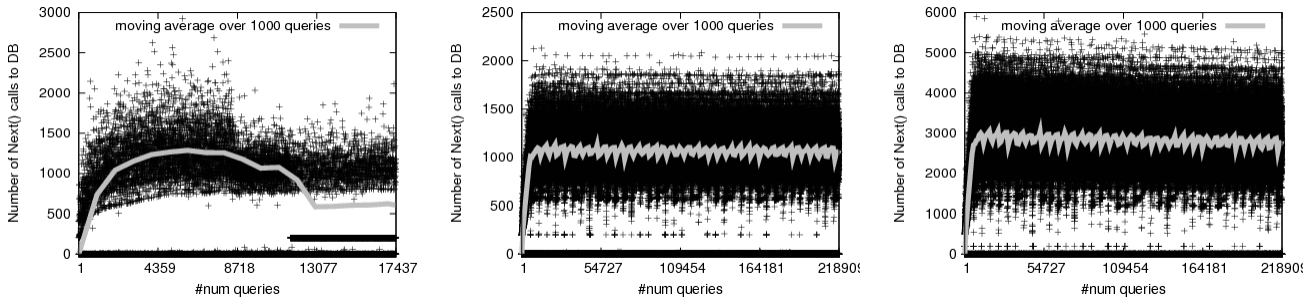
With our *LAP* approach, RT and #SA depend on the number of



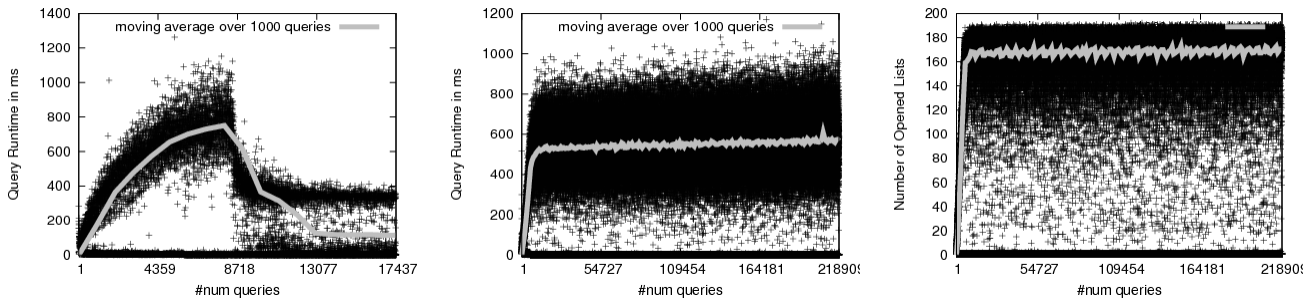
(a) EAP: RT, upd-ratio 1/100 (b) EAP: #SA, upd-ratio 1/100 (c) EAP: #MRG, upd-ratio 1/100



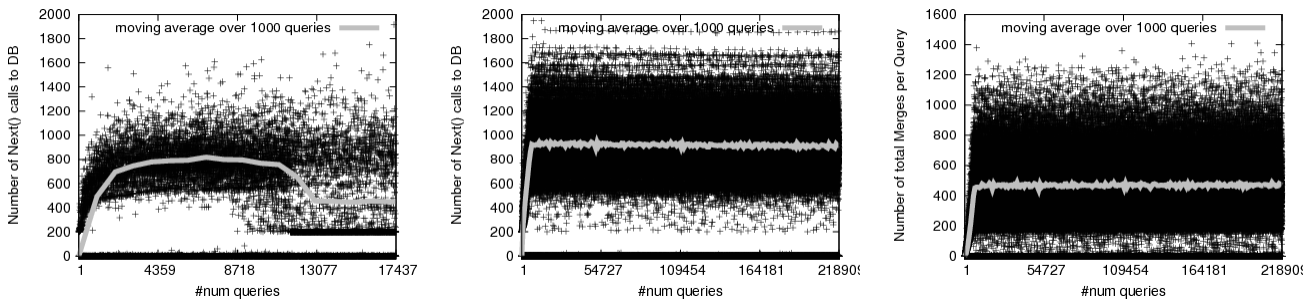
(d) EAP: RT, upd-ratio 1/1, *max-k*=200 (e) EAP: RT, upd-ratio 1/100, *max-k*=200 (f) EAP: RT, upd-ratio 1/100, *max-k*=500



(g) #EAP: SA, upd-ratio 1/1, *max-k*=200 (h) EAP: #SA, upd-ratio 1/100, *max-k*=200 (i) EAP: #SA, upd-ratio 1/100. *max-k*=500



(j) LAP: RT, upd-ratio 1/1 (k) LAP: RT, upd-ratio 1/100 (l) LAP: #OL, upd-ratio 1/100



(m) LAP: #SA, upd-ratio 1/1 (n) LAP: #SA, upd-ratio 1/100 (o) LAP: #MRGs, upd-ratio 1/100

Figure 4: *EAP and LAP on LT*: Runtime (RT) and Sequiential Accesses (#SA) measure, *top-k*=200, Update Ratio: 1 update per 1 and 100 queries.

opened lists and number of merge operations as only single list entries are merged on demand. Therefore, in the following #OL and #MRG are visualized by charts, too. Although *LAP* is able to answer queries for any number of neighbors, *top-k*=200 remains fixed in order to compare the results with *EAP*.

Results on LT (in comparison with EAP).

Figures 4j–4o show the results for *LAP* with the update ratio varying again between 1 update per 1 and 100 queries. The total number of submitted queries depends on the update ratio but is kept equal to the numbers as used with *EAP*.

From Figures 4j and 4k we can see that the average RT is slightly worse than the one for *EAP* with *max-k*=200 (but better than with *max-k*=500). However, the average #SA has improved. It also can be observed that *LAP* is more sensitive to the update ratio. For lower update ratios, both RT and #SA improves. This becomes visible in Figures 4j and 4m after 8,456 queries when all edge updates have been applied. The drop in RT and #SA results in a better performance than *EAP*. Experiments with an update ratio $1/10$ confirm this observeration (charts are ommitted for lack of space).

Figure 4n shows that #SA is fairly low, although #MRG (Figure 4o) is much higher than in *EAP* (Figure 4c) while the same number of lists have to be opened (#OL). The reason is that more MERGE operations have to be applied until all updated entries from other lists are merged in. Remember: only a single entry is read with a call of MERGE. In addition, since #SA is lower as in the *max-k*=200 case of our *EAP* approach, it also shows that indeed, in average, shorter prefixes of lists have to be read until the *top-k* nearest nodes can be correctly identified.

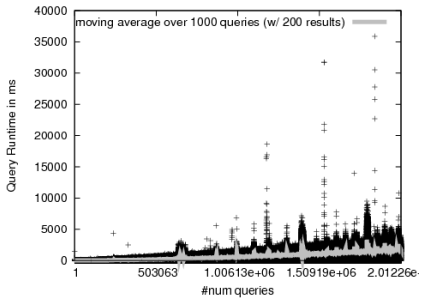
There are two reasons why the average RT per query is higher in *LAP* despite of lower #SA. First, maintaining the queue of candidates causes additional costs—again the dominating factor is I/O, not CPU time. Second, although the queue of candidates is small (no more than *top-k* nodes can be candidates), it is initialized with each query by fetching each single candidate from DB. A more efficient maintenance that allows to retrieve the lists with only one access to the DB could reduce the costs of maintenance.

Comparison From the experiments we can see that *LAP* performs equally well as *EAP* which limits queries to a fixed maximum *max-k* (better than *max-k*=500, worse than *max-k*=200). The *LAP* approach has no such limit and is able to correctly identify any amount of results and allows for queries for a variable number of neighbors. Hence, it can perfectly benefit from cases where the number of requested neighbors varies across queries. Moreover, *LAP* benefits more from time intervals with low update ratios.

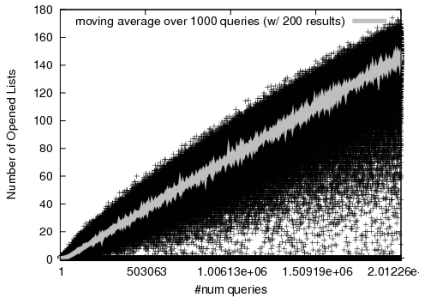
Results on Twitter.

Since the reduced Twitter graph, achieved by removing the edges for re-insertion as described in the beginning of this section, is still large and the global computation of all nearest neighbor lists by an in-memory algorithm impossible on our server, we did not precompute the transitive lists of neighbors for each user. Instead, we considered a reduced graph by randomly picking a node with 20 outgoing edges and performing a breadth-first search on the link structure of the graph. For each node found, we precomputed the list of its nearest 1000 neighbors. In this way, we precomputed the 1000 nearest neighbors for $2,427,523$ connected nodes.

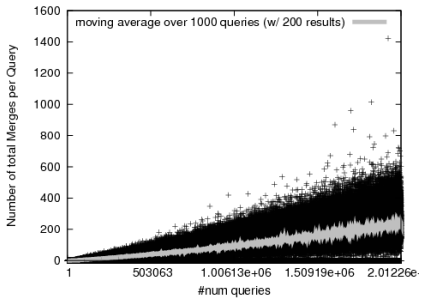
Since the number of nodes is large, it takes a huge number of queries, as shown in our experiments, until the effects of updates propagate through the graph and cause the runtimes to increase. In the begin, our algorithm has almost nothing to do and just can sequentially read the first *top-k* entries of $L(u)$ because the nearest neighbors have not yet changed. In contrast, a global recomputation of the whole graph as soon as an edge update arrives would not be feasible as no query could be correctly answered in that time and the costs for recomputing all lists are high.



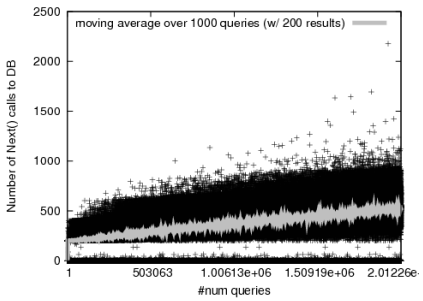
(a) RT, upd-ratio 1/100



(b) #OL, upd-ratio 1/100



(c) #MRGs, upd-ratio 1/100



(d) #SA, upd-ratio 1/100

Figure 5: *LAP Approach on Twitter*: *top-k*=200, Update Ratio: 1 update per 100 queries.

Figures 5a–5d show the result for over 2 million queries on Twitter. It can be observed that in general RT increases linearly over time with rare peaks; no such peaks can be observed for the I/O related measures #SA, #MRG, and #OL. One explanation for this odd behavior is that the experiments ran for weeks until over 2 Million queries had been processed. It is likely that these peaks in RT are

network-related. In addition, it turned out that the database needs to perform on average more than 3 accesses to the disk storage for each sequential read of a list entry. This can explain the relatively high runtimes on Twitter, even though #MRG, #OL and #SA are—even after 2 million queries—are much lower than for any other experiment on the smaller data set LT. In fact, since the I/O load on Twitter in terms of accesses etc. is actually lower than on LT, the runtime RT should be lower, too. We attribute this to the rather slow speed of the attached disk storage.

In retrospect, the choice of a relational database for storing the precomputed lists has turned out to be suboptimal. With appropriate data structures allowing to retrieve chunks of list entries at one shot, we expect to achieve much better runtimes. Further improvements could be achieved by caching lists in main memory.

7. CONCLUSIONS AND OUTLOOK

This paper presented a framework for incremental maintenance of precomputed nearest neighbors in graphs under edge insertions and decreasing edge weights. The two instantiations of the framework, *EAP* and *LAP*, can deal with large, disk-resident graphs up to millions of nodes. *EAP* can further exploit an upper bound for the number of neighbors to retrieve, while *LAP* does not need such an upper bound but achieves an equally good performance. Both approaches support a large variety of update ratios, up to extreme situations where the number of updates and queries are similar.

Our future work will consider improving the performance of the storage backend by moving to a dedicated list storage instead of a relational database. We will further attempt to evaluate our methods with realistic update loads. Furthermore, we will work towards methods for supporting increasing of edge weights and edge deletions.

8. REFERENCES

[1] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. In *STOC*, pages 117–123, 2002.

[2] P. Bouros, S. Skiadopoulos, T. Dalamagas, D. Sacharidis, and T. K. Sellis. Evaluating reachability queries over path collections. In *SSDBM*, pages 398–416, 2009.

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

[4] D. Carmel, N. Zwerdling, I. Guy, S. Ofek-Koifman, N. Har'El, I. Ronen, E. Uziel, S. Yogev, and S. Chernov. Personalized social search based on the user's social network. In *CIKM*, pages 1227–1236, 2009.

[5] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press and McGraw-Hill Book Company, 2009.

[8] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[9] C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms*, 2(4):578–601, 2006.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] G. Dong, L. Libkin, J. Su, and L. Wong. Maintaining transitive closure of graphs in SQL. *Int. Journal on Information Technology*, 5:1–23, 1999.

[12] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths in digraphs with arbitrary arc weights. *J. Algorithms*, 49(1):86–113, 2003.

[13] J. Graupmann, R. Schenkel, and G. Weikum. The SphereSearch engine for unified ranked retrieval of heterogeneous XML and web documents. In *VLDB*, pages 529–540, 2005.

[14] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large networks. In *CIKM*, pages 499–508, 2010.

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

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

[17] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.

[18] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *FOCS*, pages 81–91, 1999.

[19] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *COCOON*, pages 268–277, 2001.

[20] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[21] U. Meyer. On dynamic breadth-first search in external-memory. In S. Albers and P. Weil, editors, *STACS*, volume 1 of *LIPIcs*, pages 551–560. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008.

[22] C. Pang, G. Dong, and K. Ramamohanarao. Incremental maintenance of shortest distance and transitive closure in first-order logic and sql. *ACM Trans. Database Syst.*, 30(3):698–721, 2005.

[23] G. Ramalingam and T. W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[24] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *SIGIR*, pages 523–530, 2008.

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

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

[27] R. Xiang, J. Neville, and M. Rogati. Modeling relationship strength in online social networks. In *WWW*, pages 981–990, 2010.

[28] Y. Xiao, W. Wu, J. Pei, W. W. 0009, and Z. He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *EDBT*, pages 493–504, 2009.