TimeReach: Historical Reachability Queries on Evolving Graphs

Konstantinos Semertzidis, Kostas Lillis, Evaggelia Pitoura
Computer Science and Engineering Department
University of Ioannina, Greece
{ksemer,klillis,pitoura}@cs.uoi.gr

ABSTRACT

Since most graphs evolve over time, it is useful to be able to query their history. We consider historical reachability queries that ask for the existence of a path in some time interval in the past, either in the whole duration of the interval (conjunctive queries), or in at least one time instant in the interval (disjunctive queries). We study both alternatives of storing the full transitive closure of the evolving graph and of performing an online traversal. Then, we propose an appropriate reachability index, termed TimeReach index, that exploits the fact that most real-world graphs contain large strongly connected components. Finally, we present an experimental evaluation of all approaches, for different graph sizes, historical query types and time granularities.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems query processing

General Terms

Algorithms, Measurement, Performance

Keywords

Evolving Graphs, Historical Queries, Reachability

1. INTRODUCTION

In recent years, increasing amounts of graph structured data are being made available from a variety of sources, such as social, citation, computer and hyperlink networks. Almost all such real-world networks evolve over time, as nodes and edges are added or deleted. Analysis of their evolution finds a large spectrum of applications, ranging from social network marketing, to virus propagation and digital forensics.

In this paper, we assume that we are given an evolving set of graph snapshots corresponding to the state of the graph at different time instants. We address the problem of efficiently answering queries that involve such snapshots. In particular, we focus on a basic query type, namely reachability queries, that ask whether a node u was reachable from another node v during specific time intervals in the past. We call such queries *historical reachability queries*.

Although, there has been considerable interest in processing graph data, through a variety of graph queries including reachability, distance and pattern-based ones, querying the graph history is much less studied. The only other two approaches to building indexes for processing historical graph queries that we are aware of consider historical shortest-path queries [9, 2]. Specifically, the authors of [9] propose a method based on ordering nodes or edges pertinent to shortest path computation, while the dynamic index construction proposed in [2] does not support node or edge deletions.

All other work on historical queries focuses mainly on efficiently storing and retrieving the graph snapshots required for processing each query [14, 13, 21, 17]. In particular, in [14], a combination of graph deltas and selected materialized snapshots are explored, while in [13], the focus is on storing, sharing and processing deltas. In [21], temporally close snapshots are clustered, one representative per cluster is selected and used for an initial evaluation of the query. Finally, in [17], the placement and replication of snapshots in a distributed setting is studied. Instead, in this paper, we address the problem of building indexes for answering historical reachability queries.

Reachability queries on static graphs have been extensively studied. Research in this area follows two general directions through efficiently storing the transitive closure and speeding-up online traversal. With regards to transitive closure, various approaches have been proposed including the chain method [10, 5], methods exploring spanning trees, bit-vector compression [26] and interval [1, 28, 12], and hop [7, 22, 6] labeling. In the case of online traversal, often interval labeling [4, 25, 30] is used to prune the search space. There has also been some work on incrementally maintaining the reachability indexes in case of evolving graphs [1, 3, 23, 31], however, reachability still considers a single snapshot, i.e., the current version of the graph.

In this paper, we explore a compact representation of graph snapshots, called *version graph*, where each node and edge is annotated with the set of time intervals during which the corresponding node and edge existed in the evolving graph. We call such sets *lifespans* and seek for their minimum representation through using non-overlapping and non-continuous intervals. We also introduce a set of basic operations for efficiently manipulating lifespans of paths.

10.5441/002/edbt.2015.12

For processing historical reachability queries, we start by revisiting the basic transitive closure and online traversal approaches. For the transitive closure, we compute a minimum representation of reachability information for each pair of nodes. For the online traversal, we propose a novel interval-based traversal of the version graph along with a number of pruning steps. Furthermore, to avoid the cost and space overheads associated with precomputing the transitive closure and improving the processing cost of the online traversal, we propose a new approach, termed *TimeReach*.

TimeReach exploits the fact that most graphs consist of strongly connected components (SCCs) [20, 15]. Thus, instead of maintaining reachability information for pairs of nodes, we maintain *posting lists* with information about node membership in SCCs. We minimize the size of posting lists through an appropriate assignment of identifiers to SCCs. We show that the problem of the optimal assignment of identifiers to SCCs is equivalent to the maximum bipartite matching problem among SCCs in consequent graph snapshots. Along with postings, we maintain a condensed version graph which corresponds to the version graph of the SCCs evolution. To improve the performance of answering historical queries, we also introduce an interval-2hop approach based on pruned landmark labeling [2, 29] on the condensed version graph.

We have extensively evaluated our approach with three real social network datasets. Our experimental results show that TimeReach is space efficient, in particular for graphs consisting of large SCCs as is the case of social networks. Its incremental construction is fast; indexing a new snapshot graph takes just a few seconds. Finally, processing historical queries using TimeReach is orders of magnitude faster than the online traversal of the version graph.

The rest of this paper is structured as follows. In Section 2, we present related work, while in Section 3, we formally define historical reachability queries. In Section 4, we introduce the version graph and operations on lifespans and present the two baseline approaches, namely, the transitive closure and online traversal. In Section 5, we introduce the TimeReach Index approach, while in Section 6, we present experimental results. Section 7 concludes the paper.

2. RELATED WORK

Although, graph data management has been the focus of much current research, work in processing historical queries is rather limited. The main focus of research on evolving graphs has been on efficiently storing and retrieving graph snapshots. In this paper, our focus in on indexing for processing queries. To this end, we assume a compact representation of the sequence of graph snapshots in the form of a version graph. Alternatively, one can store just some subset of the graph snapshots in the sequence along with appropriate deltas, such that, any other snapshot can be reconstructed by applying the deltas on the selected snapshots [14, 13]. Various optimizations for reducing the storage and snapshot re-construction overheads have been proposed, such as a hierarchical index of deltas and a memory pool for the overlapping storage of snapshots [13]. Clustering temporally close snapshots and computing a representative for each cluster was also proposed [21], Deltas from representatives are stored for each cluster to achieve high compression. In the G* graph database, snapshots are efficiently stored by taking advantage of commonalities among them [16]. Different versions of each node are stored only once regardless of the number of snapshots it belongs to, and indexed by a compact in-memory index. For load balance and availability snapshot data are replicated among a number of workers.

Historical query processing in these approaches requires as a first and costly step reconstructing the relevant snapshots. Then, queries are processed through an online traversal on each of them. Query performance is addressed by trying to minimize the number of snapshots that need to be reconstructed by minimizing the number of deltas applied [14, 13], avoiding the reconstruction of all snapshots [21], or by parallel query execution and proper snapshot placement and distribution [17]. In this work, we address a different problem, that of indexing for historical reachability queries.

Historical shortest path distance queries were addressed in [9]. The authors propose a method based on ordering nodes or edges pertinent to shortest path computation. Finally, the recent work of [2] also proposes a dynamic indexing scheme for historical distance queries. However, the authors consider only insertions. This assumption simplifies the problem, since two nodes that are reachable remain reachable. The authors propose a dynamic 2hop index construction that is not applicable in the case of node or edge deletions.

Reachability queries on static graphs have been thoroughly investigated along two general directions: transitive closure compression and improving online search.

Transitive Closure Compression. Related research aims at compressing the transitive closure by storing for each node only a subset of the nodes it can reach. The first idea is to decompose the graph in k node-disjoint chains and for each node store only the first node it can reach in each chain [10, 5]. Another line of research extracts a spanning tree of the graph, and uses it to compress the transitive closure. Each node of the tree is labeled with an interval of integers such that if node u is an ancestor of v, the interval of u contains that of v. Reachability through tree edges can be easily determined by a label containment check. To incorporate reachability through non-tree edges each node inherits the intervals of its successors in the graph [1], or a partial transitive closure of non-tree edges is constructed [28]. Building upon the idea of interval labeling, a tree whose vertices are pair-wise disjoint paths extracted from the original graph is used in [12]. Another approach in compressing the transitive closure is 2-hop labeling [7, 22, 6]. Each node stores two sets of intermediate nodes: a set L_{out} of nodes it can reach and a set L_{in} of nodes that can reach it. Node u can reach node v only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$.

Speeding-up Online Traversal. These methods use interval labeling to aid online traversal by pruning the search space. In [4] and [25], a tree cover of the graph is constructed and then, for the queries that can not be answered by the tree labeling, an online search on the non-tree edges is performed using the labeling to guide the search. In [30], multiple intervals are used for the labeling. If the label containment check does not produce a negative answer, the graph is traversed online using the intervals for pruning the search.

Some of the works discuss the incremental maintenance of the index in the case of evolving graphs [1, 3, 23, 31]. However, the updated index contains reachability information only about the current version of the graph and cannot be used for answering historical queries.

The presented approaches are orthogonal to our approach in that they can be adapted so that they can be used to speed-up or avoid the online traversal of the condensed graph. We have demonstrated this by adapting, one of them, namely 2hop labeling.

3. PROBLEM DEFINITION

Most real world graphs evolve over time as new nodes or edges are added, or existing nodes or edges are deleted. We assume that time is discrete and use successive integers to denote successive time instants. There are two intuitive interpretations of time instants. One interpretation is that of actual time, for example time instant t may correspond to say October 20, 2014, 5:00am PDT. Another view is operational. In this case, time is advanced each time a graph operation, update or delete, occurs. Both interpretations of time instants are consistent with our representation.

Let $G = (V, E)$ be a directed graph where V is the set of nodes and E the set of edges. We use $G_t = (V_t, E_t)$ to denote the *graph snapshot* at time instant t, that is, the set of nodes and edges that exist at time instant t.

DEFINITION 1 (EVOLVING GRAPH). *An evolving graph $\mathcal{G}_{[t_i,t_j]}$ in time interval $[t_i, t_j]$ is a sequence $\{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$ of graph snapshots.*

An example is shown in Figure 1(a) which depicts an evolving graph $\mathcal{G}_{[t_0,t_3]}$ consisting of four graph snapshots $\{G_{t_0}, G_{t_1}, G_{t_2}, G_{t_3}\}$. For brevity, we denote time instant $t_i + 1$ as t_{i+1} and use t_i and i interchangeably, when the meaning is clear from context.

We use the term *time granularity* to refer to how often a new time instant and the corresponding graph snapshot are created. In the case of actual time, granularity may range for example from milliseconds to years, whereas, in the case of operational time, granularity may be at the level of one or more operations. A fine-grained time granularity necessitates maintaining a large amount of historical information, but supports precise historical queries.

Given a static directed graph $G = (V, E)$ and two nodes $u, v \in V$, a *reachability query* asks whether there exists a path from u to v in G. For evolving graphs, we introduce the following two types of historical reachability queries.

DEFINITION 2 (HISTORICAL REACHABILITY QUERY). *Let $\mathcal{G}_{[t_i,t_j]} = \{G_{t_i}, G_{t_i+1}, \ldots G_{t_j}\}$, be an evolving graph, $I_Q = [t_k, t_l] \subseteq [t_i, t_j]$ a time interval and v, u a pair of nodes:*

(i) a conjunctive historical reachability query $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$ returns true, if there exists a path from u to v in all graph snapshots G_{t_m}, $t_k \leq t_m \leq t_l$ of $\mathcal{G}_{[t_i,t_j]}$.

(ii) a disjunctive historical reachability query $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ returns true, if there exists a path from u to v in at least one graph snapshot G_{t_m}, $t_k \leq t_m \leq t_l$, of $\mathcal{G}_{[t_i,t_j]}$.

Our goal is to derive methods for answering reachability queries efficiently. A straightforward solution would be to build a different index for each of the graph snapshots and then pose a reachability query at each one of them. However, this solution imposes large space overheads. In addition, it requires extra processing for combining the results of each query. Instead, we propose building indexes for intervals.

4. VERSION GRAPH

In this section, we present the version graph, a natural concrete representation of an evolving graph. First, let us define the notion of lifespan. For a node u (or, edge e), its lifespan denotes the set of time intervals during which u (resp. e) existed in an evolving graph. More formally, given an evolving graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, the *lifespan*, $\mathcal{L}(u)$, (resp. $\mathcal{L}(e)$) of a node u (resp. edge e) is a set of intervals such that an interval $[t_i, t_j] \subseteq I$ belongs to $\mathcal{L}(u)$, (resp. $\mathcal{L}(e)$), if and only if, for all $t_i \leq t_m \leq t_j$, $u \in V_{t_m}$ (resp. $e \in E_{t_m}$).

We model lifespans as sets of time intervals to capture the general case of graph evolution, where nodes and edges may be deleted and then re-inserted at subsequent snapshots. Set of time intervals are also known as *temporal elements* [11]. If we do not allow deleted nodes or edges to be re-inserted, then lifespans are just intervals. Furthermore, if there are no deletions, all lifespans are intervals of the form $[t_i, t_{curr}]$, where t_i is the time instant the node or edge first appeared and t_{curr} is the time instant of the current snapshot. Therefore, in this case, lifespans can be represented simply by the time instant t_i. In the following, we use I to denote time intervals and $\mathcal{I}$ to denote sets of time intervals. To represent an evolving graph $\mathcal{G}_I$, we use a *version graph* VG_I. A version graph is a labeled directed graph that captures the evolution of the graph in a concise manner.

DEFINITION 3 (VERSION GRAPH). *Given an evolving graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, its version graph is an edge and node labeled, directed graph $VG_I = (V_I, E_I, \mathcal{L}_u, \mathcal{L}_e)$ where: $V_I = \bigcup_{t_m \in I} V_{t_m}$, $E_I = \bigcup_{t_m \in I} E_{t_m}$, $\mathcal{L}_u : V_I \rightarrow \mathcal{I}$ assigns to each node u in V_I its lifespan $\mathcal{L}_u(u)$ and $\mathcal{L}_e : E_I \rightarrow \mathcal{I}$ assigns to each edge e in E_I its lifespan $\mathcal{L}_e(e)$.*

An example is shown in Figure 1(b) which depicts the version graph for the evolving graph in Figure 1(a).

4.1 Lifespan Operations

Let us define a number of operations on lifespans, i.e., set of intervals. For two sets $\mathcal{I}$ and $\mathcal{I}'$ of time intervals, we say that $\mathcal{I}$ *covers* $\mathcal{I}'$, denoted $\mathcal{I} \sqsupseteq \mathcal{I}'$, if for each time instant t in an interval I' of $\mathcal{I}'$, there is an interval I in $\mathcal{I}$ such that t belongs to I. We also use $\mathcal{I} \sqsupseteq I$ for an interval I and $\mathcal{I} \sqsupseteq t$ for a time instant t. We say that two sets $\mathcal{I}$ and $\mathcal{I}'$ of time intervals are equivalent, $\mathcal{I} \approx \mathcal{I}'$, if $\mathcal{I} \sqsupseteq \mathcal{I}'$ and $\mathcal{I}' \sqsupseteq \mathcal{I}$.

We would like to maintain the smallest among equivalent sets of intervals. We call such sets *minimum* sets. Let us first define some simple properties for time intervals. Two time intervals $I = [t_i, t_j]$ and $I' = [t'_i, t'_j]$ are called *disjoint*, when $I \cap I' = \emptyset$ and *overlapping* otherwise. They are called *continuous* when $t'_i = t_j + 1$ and non-continuous otherwise. It is easy to see that the following proposition holds.

PROPOSITION 1.

(i) A set of intervals is minimum, if and only if, it consists of disjoint and non-continuous intervals.

(ii) For each set of time intervals, there is a unique equivalent minimum interval set.

We next define two useful operations on interval sets, namely, *join* and *merge*. Given two sets of intervals, join returns the time instants common to both, while merge returns the time instants present in at least one of them.

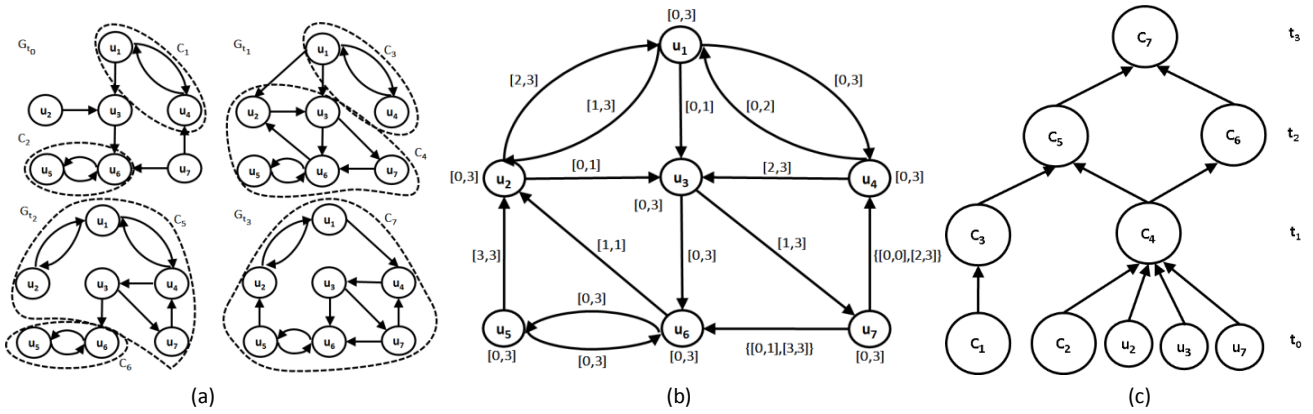

Figure 1: Example of (a) an evolving graph, (b) the corresponding version graph, (c) SCC evolution

Definition 4 (Join and Merge of Interval Sets). *Let $\mathcal{I} = \{I_1, \ldots I_k\}$ and $\mathcal{I}' = \{I'_1, \ldots I'_l\}$ be two sets of time intervals.*

(i) Join $\mathcal{I} \otimes \mathcal{I}'$ of $\mathcal{I}$ and $\mathcal{I}'$ is the minimum set equivalent to $\{I_1 \cap I'_1, \ldots I_1 \cap I'_l, \ldots, I_k \cap I'_1, \ldots I_k \cap I'_l\}$.

(ii) Merge $\mathcal{I} \oplus \mathcal{I}'$ of $\mathcal{I}$ and $\mathcal{I}'$ is the minimum set equivalent to $\mathcal{I} \cup \mathcal{I}'$.

Note that if $\mathcal{I}$ and $\mathcal{I}'$ are minimum, then the set $\{I_1 \cap I'_1, \ldots I_1 \cap I'_l \ldots, I_k \cap I'_1 \}$ is a minimum set, whereas the set $\{I_1 \cup I'_1, \ldots I_1 \cup I'_l, \ldots, I_k \cup I'_1 \ldots I_k \cup I'_l\}$ may not be minimum.

The lifespan $\mathcal{L}(p)$ of a path p includes the time intervals during which all its edges coexist. Clearly, for a path $p = e_1 \ldots e_m$, it holds that $\mathcal{L}(p) = \mathcal{L}_e(e_1) \otimes \ldots \otimes \mathcal{L}_e(e_m)$, where $\mathcal{L}_e(e_i)$, $1 \leq i \leq m$, is the lifespan of e_i. For example, for path $p = ((u_4, u_3), (u_3, u_7), (u_7, u_6))$ in Figure 1(b), $\mathcal{L}(p)$ = {[2,3]} $\otimes$ {[1,3]} $\otimes$ {[0,1], [3, 3]} = {[3,3]}, while for path p'= $((u_1, u_3), (u_3,u_7), (u_7,u_4))$, $\mathcal{L}(p')$ = {[0,1]} $\otimes$ {[1,3]} $\otimes$ {[0,0], [2,3]} = $\emptyset$.

We can now define the lifespan, $\mathcal{L}(u, v)$, of the reachability between two nodes u and v. Let $P(u, v) = \{p_1, \ldots p_l\}$ be the set of all paths from u to v. $\mathcal{L}(u, v)$ depends on the lifespans of all possible paths in VG_I from u to v, in particular, $\mathcal{L}(u, v) = \mathcal{L}(p_1) \oplus \ldots \oplus \mathcal{L}(p_l)$. For example, for nodes u_4 and u_6 in Figure 1(b), $P(u_4, u_6) = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ where $p_1 = u_4u_3u_6$, $p_2 = u_4u_3u_7u_6$, $p_3 = u_4u_1u_3u_6$, $p_4 = u_4u_1u_3u_7u_6$, $p_5 = u_4u_1u_2u_3u_6$, $p_6 = u_4u_1u_2u_3u_7u_6$ (note, that for notational brevity, paths were denoted by the participating nodes instead of edges). Then, $\mathcal{L}(u_4, u_6) = \{[2, 3]\}$ $\oplus$ $\{[3, 3]\}$ $\oplus$ $\{[0, 1]\}$ $\oplus$ $\{[1, 1]\}$ $\oplus$ $\{[1, 1]\}$ $\oplus$ $\{[1, 1]\}$ = $\{[0, 3]\}$.

Clearly, historical reachability queries can be represented in terms of lifespans. Specifically, given a version graph VG_I, a time interval $I_Q = [t_k, t_l] \subseteq [t_i, t_j]$ and two nodes v, u,

(i) a conjunctive historical reachability query $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$ returns true, if and only if, $\{I_Q\} \otimes \mathcal{L}(u, v) \sqsupseteq I_Q$.

(ii) a disjunctive historical reachability query $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ returns true, if and only if, $\{I_Q\} \otimes \mathcal{L}(u, v) \neq \emptyset$.

To represent lifespans, we use bit arrays. Assume without loss of generality, that the maximum time instant, that is, the number of graph snapshots, is T. Then, a lifespan, i.e., set of intervals, $\mathcal{I}$ is represented by a bit array B of size T, such that $B[i] = 1$ if $\mathcal{I} \sqsupseteq i$, and 0, otherwise. For example, take $\mathcal{I} = \{[2, 4], [9, 10], [13, curr]\}$ and $T = 16$. The bit array representation of $\mathcal{I}$ is 00111000011001111. This leads to an efficient implementation of both join $\otimes$ and merge $\oplus$. In particular, let $\mathcal{I}$ and $\mathcal{I}'$ be two set of intervals and B and B' be their bit arrays. Then, $\mathcal{I} \otimes \mathcal{I}'$ is computed as B logical-AND B' and $\mathcal{I} \oplus \mathcal{I}'$ as B logical-OR B'. An alternative representation would be to use ordered lists of intervals. Lifespan operations would then be performed using variations of merge sort resulting in $O(T)$ complexity. Lists impose in general large computational overheads in computing reachability.

4.2 Baseline Approaches

There are two baseline approaches to answering reachability queries on static graphs, namely pre-computation of the graph transitive closure and online traversal of the graph. In this section, we revisit these baseline approaches for historical reachability queries on a version graph.

4.2.1 Historical Transitive Closure

Instead of maintaining a different transitive closure for each graph snapshot of the evolving graph $\mathcal{G}_I$, we maintain a single transitive closure, CL_I for the version graph VG_I. The transitive closure includes for each pair of nodes u, v, their reachability lifespan, $\mathcal{L}(u, v)$. To construct the transitive closure, we use a variation of the Floyd-Warshall algorithm that takes into account lifespans, shown in Algorithm 1. If there is a path $p_{u,w}$ from node u to node w and a path $p_{w,v}$ from node w to node v then there exists a path $p_{u,v} = (p_{u,w}, p_{w,v})$ from u to v with $\mathcal{L}(p_{u,v}) = \mathcal{L}(p_{u,w}) \otimes \mathcal{L}(p_{w,v})$ and $\mathcal{L}(p_{u,v})$ is merged with the $\mathcal{L}(u, v)$ computed so far.

The time complexity for Algorithm 1 is $O(|V_I|^3T)$ in the worst case and requires storage in the order of $|V_I|^2$. For answering a reachability query $u \overset{I_{Q\vee}}{\rightsquigarrow} v$ or $u \overset{I_{Q\wedge}}{\rightsquigarrow} v$, initially the entry $\mathcal{L}(u, v)$ in CL_I is located and then joined with the query interval I_Q, thus requiring constant time complexity.

4.2.2 Online Traversal of the Version Graph

A straightforward approach to process a reachability query for an interval I_Q would be to perform an online traversal on all graph snapshots G_t, $t \in I_Q$. When using the version graph representation, this corresponds to traversing

Algorithm 1 TransitiveClosure(VG_I)

Input: Version graph VG_I
Output: The transitive closure CL_I

```
1: for all u, v ∈ V_I × V_I do
2:    if (u, v) ∈ E_I then
3:       CL_I(u, v) = L_e((u, v))
4:    else
5:       CL_I(u, v) = ∅
6:    end if
7: end for
8: for w = 1 to |V_I| do
9:    for all u, v ∈ V_I × V_I do
10:      CL_I(u, v) = CL_I(u, v) ⊕ (CL_I(u, w) ⊗ CL_I(w, v))
11:   end for
12: end for
```

only edges e such that $\mathcal{L}_e(e) \sqsupseteq t$, once for each $t \in I_Q$. We call this approach, *instant based traversal*.

To avoid multiple traversals, i.e., one for each snapshot in I_Q, we consider an *interval based traversal* of the version graph. The BFS-based interval traversal for disjunctive historical queries is shown in Algorithm 2 and for conjunctive historical queries in Algorithm 3.

In particular, for conjunctive queries, since a node v may be reachable from u through different paths at different graph snapshots, we maintain an interval set $\mathcal{R}$ with the part of $\mathcal{L}(u, v) \otimes I_Q$ covered so far (line 9, Algorithm 3). The traversal ends when $\mathcal{R}$ covers the whole query time interval I_Q (line 10, Algorithm 3).

To speed-up traversal, we perform a number of pruning tests. The traversal stops when we reach a node whose lifespan is outside the query interval. In addition, the traversal stops at a neighbor w of a node n when $\{I_Q\} \otimes \mathcal{L}_e(n, w) \neq \emptyset$ since a node v cannot be reachable through an edge which is not alive in at least one t inside the query interval (line 6, Algorithms 2 and 3).

Still an edge may be traversed multiple times, if it participates in multiple paths from source to target. To reduce the number of such traversals, we provide additional pruning by recording for each node w, an interval set $\mathcal{IN}(w)$ with the parts of the query interval for which it has already been traversed. If the query reaches w again looking for interval I' $\subseteq I_Q$ and $\mathcal{IN}(w) \sqsupseteq I'$, the traversal is pruned (line 11 of Algorithm 2, line 15 of Algorithm 3).

For example, consider the version graph in Figure 1(b) and query $u_1 \overset{[0,3]\wedge}{\rightsquigarrow} u_5$. Paths $p_1 = u_1u_3u_6u_5$, $p_2 = u_1u_3u_7u_6u_5$, $p_3 = u_1u_2u_3u_6u_5$, $p_4 = u_1u_2u_3u_7u_6u_5$, $p_5 = u_1u_4u_3u_6u_5$ and $p_6 = u_1u_4u_3u_7u_6u_5$ with $\mathcal{L}(p_1) = \{[0, 1]\}$, $\mathcal{L}(p_2) = \{[1, 1]\}$, $\mathcal{L}(p_3) = \{[1, 1]\}$, $\mathcal{L}(p_4) = \{[1, 1]\}$, $\mathcal{L}(p_5) = \{[2, 3]\}$ and $\mathcal{L}(p_6)$ $= \{[3, 3]\}$ need to be traversed to conclude correctly that the result of the query is true. Hence, some edges, e.g., (u_3, u_6), (u_6, u_5) need to be traversed multiple times for different time intervals $I'_i \subseteq I_Q$. However, when the query reaches u_3 again through path p_3, it is pruned and it does not traverse the edge (u_3, u_6) since $\mathcal{IN}(u_3)$ is equal to {[0,1]} which covers the current query interval $I' = \{[1,1]\}$.

Since in the worst case for both instant and interval based traversal each edge may be traversed $|I_Q|$ times, the complexity for both traversals is $O((|V_I| + |E_I|)|I_Q|)$. However, in practice interval based traversal outperforms the instant based one since each edge traversal covers large parts of the query interval instead of a single time instant. Furthermore, pruning guarantees that an edge will not be traversed twice for the same interval.

Algorithm 2 Disjunctive-BFS(VG_I, u, v, $\{I_Q\}$)

Input: Version graph VG_I, nodes u, v, interval $I_Q \subseteq I$
Output: True if v is reachable from u in any time instant in I_Q and false otherwise

```
1: create a queue N, create a queue INT
2: enqueue u onto N, enqueue I_Q onto INT
3: while N ≠ ∅ do
4:    n ← N.dequeue()
5:    i ← INT.dequeue()
6:    for all w s.t. (n, w) in VG_I and {I_Q} ⊗ L_e((n, w))
      ≠ ∅ do
7:       if w == v then
8:          Return(true)
9:       end if
10:      I' = {I_Q} ⊗ L_e(u, w)
11:      if IN(w) ⋢ I' then
12:         IN(w) = IN(w) ⊕ I'
13:         enqueue w onto N
14:         enqueue I' onto INT
15:      end if
16:   end for
17: end while
18: Return(false)
```

5. THE TIMEREACH INDEX

Our approach exploits the fact that many real-world social graphs are characterized by large strongly connected components (SCC) [20, 15]. Thus, instead of maintaining reachability information for pairs of nodes, we maintain information about the SCCs that each node belongs to. If two nodes belong to the same component, then they are reachable. However, as the graph evolves over time, its strongly connected components change as well. An example is shown in Figure 1(c) that depicts the SCCs of the graph in Figure 1(b) as they evolve over time.

Given an evolving graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, we invoke at each graph snapshot $G_{t_k} \in \mathcal{G}_I$ an algorithm, e.g., Tarjan's algorithm [24], to identify the corresponding set of SCCs. A unique id is assigned to each SCC at each snapshot.

For each node u, we maintain a list $P(u)$ that contains (C, t) pairs specifying the strongly connected component C to which node u belongs at time instant t. $P(u)$ is called *posting list* and each pair in the list a *posting*. The storage complexity is $\Omega(|V_I||I|)$, since each node participates in at most one SCC at each time instant. If we use Tarjan's algorithm [24], the time complexity for constructing the lists is $O((|V_I| + |E_I|)|I|)$, since each run of the Tarjan's algorithm has an $O(|V_I| + |E_I|)$ complexity.

For presentation clarity, we assume that single nodes form singleton SCCs whose ids are the ids of the corresponding nodes. However, for space efficiency, we do not maintain postings in this case.

We perform an additional optimization. Many nodes have strong connections, i.e. they remain in the same components even in the face of component splits and joins. We exploit this fact to reduce the storage space required for the postings by observing that the posting lists of these nodes consist of

Algorithm 3 Conjunctive-BFS(VG_I, u, v, $\{I_Q\}$)

Input: Version graph VG_I, nodes u, v, interval $I_Q \subseteq I$
Output: True if v is reachable from u in all time instants in I_Q and false otherwise

```
1: create a queue N, create a queue INT
2: enqueue u onto N, enqueue I_Q onto INT
3: while N ≠ ∅ do
4:    n ← N.dequeue()
5:    i ← INT.dequeue()
6:    for all w s.t. (n, w) in VG_I and {I_Q} ⊗ L_e((n,w)) ≠ ∅ do
7:       I' = {I_Q} ⊗ L_e(n,w)
8:       if w == v then
9:          R = R ⊕ I'
10:         if R ⊒ I_Q then
11:            Return(true)
12:         end if
13:         continue
14:      end if
15:      if IN(w) ⋣ I' then
16:         IN(w) = IN(w) ⊕ I'
17:         enqueue w onto N
18:         enqueue I' onto INT
19:      end if
20:   end for
21: end while
22: Return(false)
```

the same elements. We avoid redundancy by storing such lists only once and replacing the posting lists of the relevant nodes with pointers to the common list. We call this approach *posting sharing*.

An example is shown in Figure 2(a), where, for instance, the first posting list indicates that nodes with ids 1 up to 50 belong to the strongly connected component with id C_1 at time t_0, C_6 at t_1 and C_9 at t_2.

In addition, for each graph snapshot G_{t_k}, we construct a SCC graph snapshot $G_{S_{t_k}} = (V_{S_{t_k}}, E_{S_{t_k}})$ such that there is a node U in $V_{S_{t_k}}$ for each SCC in G_{t_k}, and there is an edge (U, V) in $E_{S_{t_k}}$, if and only if, there is an edge (u, v) in G_{t_k} from a node u that belongs to the SCC that corresponds to U to a node v that belongs to the SCC that corresponds to V. For a time interval $I = [t_i, t_j]$, this results in an evolving SCC graph $\mathcal{G}_{S_I} = \{G_{S_{t_i}}, G_{S_{t_i+1}}, \ldots, G_{S_{t_j}}\}$. We construct the SCC graphs incrementally, as the SCCs are created. The size of each SCC graph depends on the size of the original snapshot graph and in the worst case is equal to it.

We call this approach simple *TimeReach* (TR). To answer a reachability query $u \overset{I_Q^{\wedge}}{\rightsquigarrow} v$, (or, $u \overset{I_Q^{\vee}}{\rightsquigarrow} v$), we check for each $t \in I_Q$ whether u and v belong to the same component. If this is not the case, we traverse the corresponding G_{S_t}.

Next, we present a more space efficient method of exploiting strongly connected components for historical queries.

5.1 Condensed TimeReach

While in the TR approach, we maintain information per time instant, we would like to aggregate such information to express SCC participations during time intervals. In this case, a posting (C, I'), $I' \subseteq I$, belongs to $P(u)$, if u participates in the SCC with id C at all time instants in I'. Our goal is to minimize the total number of such postings.

Problem 1 (Optimal SCC-id assignment). *Given a time interval I and a set of SCCs for each $t \in I$, find an assignment of ids to SCCs that results in the minimum number of postings.*

A new posting is created, each time a node participates in a different SCC. Thus, SCC ids should be re-assigned so that the number of such new postings is minimized. We use a weighted graph to formalize the optimal assignment of ids to SCCs.

In particular, we model SCC evolution over a time interval I using a weighted graph $G_C(V_C, E_C, \mathcal{W})$ where each node $U \in V_C$ corresponds to a SCC that existed at some time instant $t \in I$, and an edge $e = (U, V) \in E_C$, if and only if, SCC U existed at time t_k, SCC V existed at time $t_k + 1$ and there is at least one node that belongs to both U and V. $\mathcal{W}$ assigns to each edge $e = (U, V)$ a weight $\mathcal{W}(e)$ that corresponds to the number of nodes that belong to both U and V.

An example of a weighted graph is shown in Figure 2(b) that depicts the evolution of the graph whose posting lists are shown in Figure 2(a). For instance, component C_7 created at time instant t_1 consists of 100 nodes from component C_4 and 150 nodes from C_5.

Let $G_{C_{[t_k,t_k+1]}}(V_{C_{[t_k,t_k+1]}}, E_{C_{[t_k,t_k+1]}}, \mathcal{W})$ be the subgraph of $G_C(V_C, E_C, w)$, that consists of the nodes $U \in V_{C_{[t_k,t_k+1]}}$ that correspond to the SCCs that exist at time interval $[t_k, t_k + 1]$. $G_{C_{[t_k,t_k+1]}}$ represents one step in the SCC evolution. Note that, from the definition of G_C, $G_{C_{[t_k,t_k+1]}}$ is a bipartite graph.

We make the following observation. At time instant $t_k + 1$, a new posting is created exactly for those nodes that participated in a different SCC at $t_k + 1$ than at t_k. The number of these new postings is equal to the sum of weights from node U to V in $G_{C_{[t_k,t_k+1]}}$ where U has a different id than V. Thus, to minimize the number of new postings, we have to maximize the weight of the edges between pairs of nodes that have the same id. This corresponds to finding a maximum bipartite matching of $G_{C_{[t_k,t_k+1]}}$.

Theorem 1. *The optimal SCC-id assignment problem can be reduced to the problem of finding the maximum weight bipartite matching (MWM) M_k of each $G_{C_{[t_k,t_k+1]}}$.*

Proof. As shown above, solving the MWM for each bipartite graph $G_{C_{[t_k,t_k+1]}}$ minimizes the number of new postings created at $t_k + 1$. We shall show that this step-wise assignment is optimal overall in G_C. For the purposes of contradiction, assume that the optimal assignment is a set N of edges, $N \subset E_C$ and that N is different from the set of edges attained through the maximum bipartite matchings, that is, $\sum_{e \in N} w(e) > \sum_k \sum_{e \in M_k} w(e)$. Hence, for some m, for $N_m = N \cap E_{C_{[t_m,t_m+1]}}$ it holds that $\sum_{e \in N_m} w(e) > \sum_{e \in M_m} w(e)$, which means that M_m is not a MWM, which is a contradiction. □

Figure 2(c) shows the weighted graph after the assignment of new ids through bipartite matching, while Figure 2(d) shows the new posting lists.

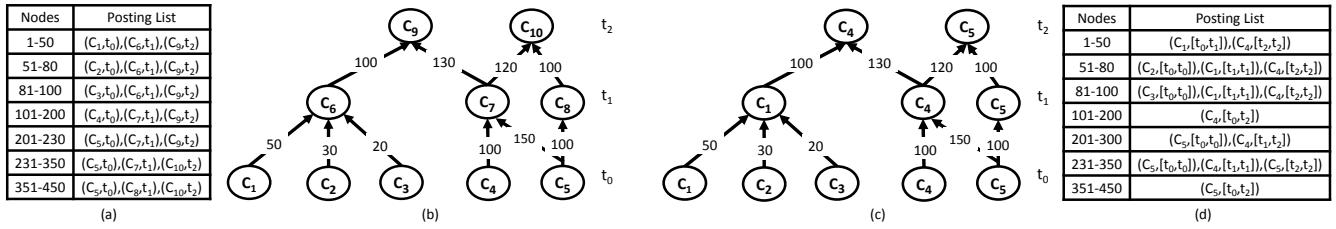

Figure 2: (a) Shared posting lists, (b) weighted graph modeling the evolution of SCCs, (c) weighted graph after the bipartite matching, and (d) the compressed shared posting lists

The maximum weight bipartite matching problem is well-studied (e.g., see [8] for a survey). The most widely used algorithm for solving this problem on a graph $G(V, E)$ is the Hungarian algorithm whose running time ranges from $O(|V|^3)$ to $O(|E||V| + |V|^2 loglog|V|)$ depending on the implementation. Another category of algorithms depends on the edge weights and the fastest one runs in $O(|E|\sqrt{|V|}logW)$ time, where W is the maximum edge weight. In addition, a number of fast approximation algorithms have been proposed. The simplest such algorithm is the greedy algorithm that sorts the edges by weight and repeatedly picks the edge with the largest weight. This algorithm can be implemented with $O(|E|)$ time complexity and produces a 1/2 worst case approximation.

The incremental algorithm for constructing the SCC postings is presented in Algorithm 4. It takes as input the current snapshot and the postings computed up to the previous snapshot, and constructs the current postings. It starts by computing the SCCs using Tarjan's algorithm with complexity $O(|V_t| + |E_t|)$ (line 2). Then, it constructs the graph $G_{C_{[t,t+1]}}$ with complexity $O(|E_{C_{[t-1,t]}}|)$ (line 5). Next, the MWM is computed and new ids are assigned to the new SCCs (lines 6 - 9). The complexity of this step depends on which algorithm is used for computing the MWM. We use the greedy algorithm with complexity $O(|E_{S_{[t-1,t]}}|)$. Finally, the SCC postings are created/updated for each node of the current snapshot, creating a new entry only for nodes that participate in a different SCC (with a different id) than the one in time instant $t-1$ (lines 11 - 22). The complexity of these steps is $O(|V_t|)$ since each operation in the loop has constant time complexity. Thus, in total the running time of the algorithm is $O(|V_t| + |E_t|)$.

As in the simple TR approach, we also construct the evolving SCC graph, which in this case has a much smaller number of nodes due to the reduction of the number of strongly connected components achieved by the bipartite matching.

Finally, we construct the version graph $VG_{S_I} = (V_{S_I}, E_{S_I}, \mathcal{L}_u, \mathcal{L}_e)$ of the evolving SCC graph that we call *condensed version graph*. We construct the condensed version graph incrementally as follows. For each snapshot $G_{t_i} \in \mathcal{G}_I$, for each edge $(u, v) \in E_{t_i}$ we look up the postings $P(u)$, $P(v)$ for entries (U, I'), (V, I'') s.t. $t_i \in I'$ and $t_i \in I''$. If $U \neq V$ and edge $(U, V) \notin E_{S_I}$, the edge is added with lifespan $\{[t_i, t_i]\}$, otherwise the lifespan of the edge is extended to include t_i. We call the above approach *condensed TimeReach* (TRC).

5.2 Query Processing

Query processing of a (disjunctive or conjunctive) reachability query $u \overset{I_Q}{\rightsquigarrow} v$ is performed in two steps. In the first step, the appropriate postings of nodes u and v are retrieved. If the two nodes belong to the same strongly connected component during the whole query interval for conjunctive queries or once for disjunctive queries, the answer is true. Otherwise, let $\mathcal{I}'_\mathcal{Q}$ be the set of intervals during which nodes u and v belong to different components. The query is re-written as a set of reachability sub-queries of the form $U_k \overset{I_{Q_i}}{\rightsquigarrow} V_m$, where u belongs to SCC U_k and v belongs to SCC V_m for some common time interval I_{Q_i}, $\mathcal{I}'_\mathcal{Q} \sqsupseteq I_{Q_i}$, the set $\mathcal{I}_\mathcal{Q} = \bigcup_i I_{Q_i}$ consists of disjoint intervals, and $\mathcal{I}_\mathcal{Q} \approx \mathcal{I}'_\mathcal{Q}$. The results of the sub-queries are combined to produce the answer for the query through an AND (OR) for conjunctive (disjunctive) queries.

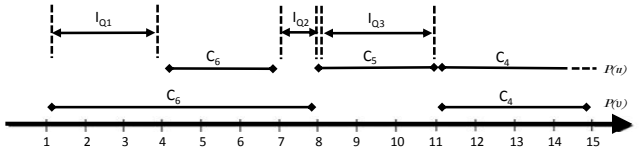

Figure 3: Example of splitting query $u \overset{[1,15]\wedge}{\rightsquigarrow} v$

For example, consider the query $u \overset{[1,15]\wedge}{\rightsquigarrow} v$ in Figure 3, where the posting lists for u and v are respectively, $P(u) = (C_6\ [4, 7],\ C_5\ [8, 11],\ C_4\ [11, curr]$ and $P(v) = (C_6\ [1, 8],\ C_4\ [11, 15])$. The query is split in three sub-queries: $u \overset{I_{Q_1}}{\rightsquigarrow}\wedge\ C_6$, $u \overset{I_{Q_2}}{\rightsquigarrow}\wedge\ C_6$, $v \overset{I_{Q_3}}{\rightsquigarrow}\wedge\ C_5$.

In the worst case, the two nodes belong to a different SCCs at each time instant in I_Q, thus we need to traverse the condensed version graph for each t with a cost of $O(|I_Q|(|V_{S_I}| + |E_{S_I}|))$ Two factors that influence performance are the number of postings for each node and the size of the condensed version graph. The smaller the number of postings, the fewer sub-queries are required in the second step. The smaller the condensed version graph, the faster the traversals. Hence, the optimal assignment of SCC ids is crucial to query processing performance, since it keeps the posting lists short and the size of the condensed version graph small.

5.3 Interval 2Hop

Reachability on version graphs can be made more efficient by maintaining additional information. In this paper, we use an approach based on pruned landmark 2hop labeling [2, 29]. The idea is that for each node u of a given graph, we maintain two labels $L_{in}(u)$ and $L_{out}(u)$ which include nodes that can reach u and can be reached by u respectively. The labels are computed such that a node u reaches v, if an only if, $L_{in}(v) \cap L_{out}(u) \neq \emptyset$. Instead of traversing the graph, a reachability query can now be answered by using the labels.

For historical reachability queries, we also keep along with each node w in $L_{in}(v)$ the reachability lifespan $\mathcal{L}(w, v)$ and along with each node w in $L_{out}(u)$ the reachability lifespan

Algorithm 4 ConstructSccPostings(G_t, P_{t-1}, $G_{S_{[t-2,t-1]}}$)

Input: Snapshot G_t, SCC postings P_{t-1}
Output: SCC postings P_t

1: $S_{SCC_t} = \emptyset$, $M = \emptyset$
2: Run Tarjan's algorithm on G_t
3: S_{SCC_t} is the set of the detected SCCs where each $SCC_i \in S_{SCC_t}$ is assigned a unique id C_i
4: **if** $t > 0$ **then**
5: Construct $G_{S_{[t-1,t]}}$ from S_{SCC_t} and $G_{S_{[t-2,t-1]}}$
6: Compute maximum weight matching M
7: **for all** edges $e = (U, V) \in M$ **do**
8: $C_v = C_u$
9: **end for**
10: **end if**
11: **for all** nodes $u \in V_t$ **do**
12: find $SCC_i \in S_{SCC_t}$ s.t. $u \in SCC_i$
13: **if** $P_{t-1}(u) \neq \emptyset$ **then**
14: **if** $P_{t-1}(u)[end].C \neq C_i$ **then**
15: $P_{t-1}(u)[end].I = [t_s, t-1]$
16: $P_{t-1}(u).add(C_i, [t, curr])$
17: **end if**
18: **else**
19: $P_{t-1}(u).add(C_i, [t, curr])$
20: **end if**
21: **end for**
22: $P_t = P_{t-1}$

$\mathcal{L}(u, w)$. In the presence of 2hop labels, to answer a query $u \overset{I_Q^\wedge}{\rightsquigarrow} v$ ($u \overset{I_Q^\vee}{\rightsquigarrow} v$), we compute the set $L_{in}(v) \cap L_{out}(u)$ and then for each w in $L_{in}(v) \cap L_{out}$, we join the lifespan of w in $L_{in}(v)$ with the lifespan of w in $L_{out}(u)$. To answer the query the joined lifespans $\mathcal{L}$ (w) of nodes w in $L_{in}(v) \cap L_{out}$ are joined with the query interval $\mathcal{L}$ to see whether they cover I_Q (or, have at least a time instant in common).

We compute the labels for the nodes of the condensed version graph, incrementally. For an interval $I = [t_i, t_j]$, we compute the labels for the SCC graph snapshots at each time t in I, starting from t_i. For each time t_k, $t_k > t_i$, we merge the labels computed for a node C at time t_k, with the labels computed for C at the previous time $t_k - 1$. For the construction of L_{in} and L_{out} for each SCC graph snapshot at time instant t_k, we process the nodes of the graph by using the $INOUT$ strategy that starts a BFS traversal from the nodes with the largest $(indegree(u)+1) \times (outdegree(u)+1)$ [29]. An example of the final 2hop labels of each SCC node in a version graph is given in Figure 4.

6. EXPERIMENTAL EVALUATION

To evaluate our approach, we used three real datasets: Facebook (FB) [27], Flickr (FL) [19] and YouTube (YT) [18]. The characteristics of each dataset are shown in Table 3. For example, FB consists of 871 daily snapshots of the New Orleans Facebook friendship graph, which correspond to 125 weekly or 29 monthly snapshots. We report the number of nodes, edges, and SCCs (singleton SCCs are not included) and the size of the largest SCC at the first and last snapshot.

All three datasets are treated as directed. Also, all datasets are insert-only, i.e. they do not contain information about node/edge deletions. Therefore, we synthetically generate random edge deletes. The input parameters and their default values are shown in Table 1.

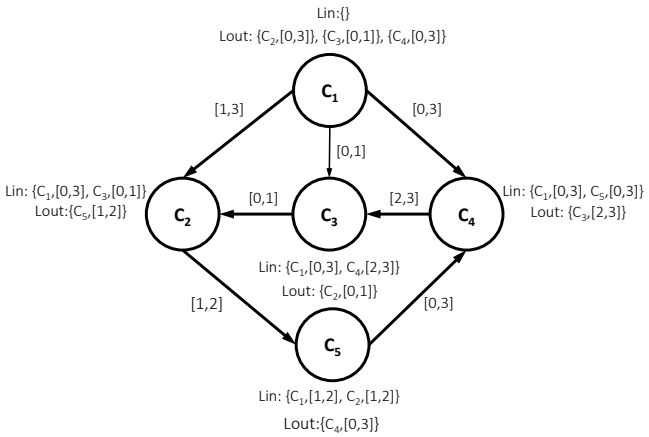

Figure 4: An example of interval 2hop labels

We evaluate the size and the construction time of the Version Graph (VG), the Transitive Closure (TC), the simple TimeReach (TR), the condensed TimeReach (TRC) and the condensed TimeReach with 2hop labels (TRCH). We also evaluate the online processing of historical reachability queries using an instant-based (INS) or interval-based (INT) traversal of the version graph and using the various TimeReach indexes. Table 2 summarizes the various approaches.

We ran our experiments on a system with a quad-core Intel Core i7-3820 3.6 Ghz processor and 64 GB memory. We only used one core for all experiments.

Table 1: Input parameters

		# of nodes	Snapshot granularity	Query interval (in days)	% of deletes
FB	Default	61,096	day	7	10
	Range	117 - 61,096	day, week, month	7 - 35	0 - 30
YT	Default	1,138,499	day	7	10
	Range	1,004,777 - 1,138,499	day, week, month	7 - 35	0 - 30
FL	Default	2,302,925	day	7	10
	Range	1,487,058 - 2,302,925	day, week, month	7 - 35	0 - 30

6.1 Index Size

In the first set of experiments, we evaluate the various approaches in terms of their storage requirements. The size of the TR and TRC include the storage required for maintaining the posting lists and the SCC graphs, while the size of the TRCH includes in addition the storage required for the 2hop labels.

Table 2: Overview of difference approaches

VG	Version Graph
TC	Transitive Closure
TR	(Simple) TimeReach
TRC	Condensed TimeReach
TRCH	Condensed TimeReach with 2hop labels
INS	Instant-based traversal of the version graph
INT	Interval-based traversal of the version graph

Table 3: Dataset properties

	Snapshot Granularity	# nodes		# edges		# SCC		Max SCC (# nodes)	
		first	last	first	last	first	last	first	last
FB	(daily) 871	117	61,096	128	1,139,081	10	374	3	51,286
	(weekly) 125	1,429	61,096	2,365	1,139,081	138	374	18	51,286
	(monthly) 29	4,239	61,096	12,224	1,139,081	279	374	96	51,286
YT	(daily) 37	1,004,777	1,138,499	4,379,283	4,452,646	9,807	11,360	457,932	509,332
	(weekly) 6	1,025,536	1,138,499	4,379,283	4,452,646	9,807	11,360	465,668	509,332
	(monthly) 2	1,116,602	1,138,499	4,446,042	4,452,646	10,664	11,360	485,273	509,332
FL	(daily) 134	1,487,058	2,302,925	17,022,083	33,140,018	42,163	58,636	1,004,426	1,605,184
	(weekly) 20	1,507,700	2,302,925	17,393,321	33,140,018	42,163	58,636	1,010,498	1,605,184
	(monthly) 5	1,585,173	2,302,925	18,987,847	33,140,018	42,459	58,636	1,081,499	1,605,184

Graph Size (scalability). Figure 6 reports the size for varying number of nodes. As shown, TRC is much smaller than TR in all cases. For FB and FL, the largest SCC covers 83% and 70% of the graph respectively, while for YT, it covers just 45% (see Table 3). Thus, the TRC size for the FB dataset is 89% smaller, while for the YT and FL datasets, we achieve 40% and 57% of compression respectively. The larger the SCCs, the higher the compression achieved.

Since the size of the transitive closure (TC) grows rapidly, we compute TC for a smaller subset of the FB dataset varying the number of nodes from 1,000 to 6,000. As shown in Table 4, even for this small graph, the size of TC reaches 106 MB.

Percentage of Deletes. For each dataset, we vary the percentage of edge deletes from 0% to 30% of edge insertions. Table 5 presents the results for the FB dataset. We observe that the size of TR and TRC decreases; this can be explained by the fact that deletions affect the isolated nodes that become disconnected from the components and thus there are less edges between components and isolated nodes. The size of VG remains constant, since the size of the lifespan labels remains the same. Finally, the size of TRCH increases, because in case of deletes, additional nodes need to be included in the 2hop labels for ensuring the reachability test.

Table 4: Comparison with transitive closure

# nodes	Size (MB)			Constr. Time (sec)		
	TR	TRC	TC	TR	TRC	TC
1,000	0.013	0.012	2.91	0.01	4.76	167.49
2,000	0.026	0.009	11.56	0.23	5.02	1,457
3,000	0.039	0.012	26.27	0.35	5.89	5,788
4,000	0.052	0.018	47.12	0.41	6.33	16,580
5,000	0.063	0.026	73.97	0.59	6.79	39,112
6,000	0.074	0.032	106.82	0.72	7.13	81,123

Snapshot Granularity. Table 6 reports the storage required for maintaining daily, weekly and monthly snapshots of the three datasets. All sizes increase with the number of snapshots. For example, for FL, the increase of the number of snapshots by a factor of 30 (from 5 monthly to 134 daily) causes an increase of the size of TR by a factor of 3.44. The size of TR and TRC decreases with the snapshot granularity (number of snapshots) since less snapshots mean less postings and smaller SCC graphs. The size of VG does not decrease significantly, because it requires memory to keep lifespan labels for all nodes and edges of the graph.

Table 5: Size per % of deletes (Facebook)

% of deletes	Size (MB)			
	VG	TR	TRC	TRCH
0	11	0.5	0.21	1,493
10	11	0.58	0.22	1,528
20	11	0.45	0.19	1,612
30	11	0.47	0.18	1,664

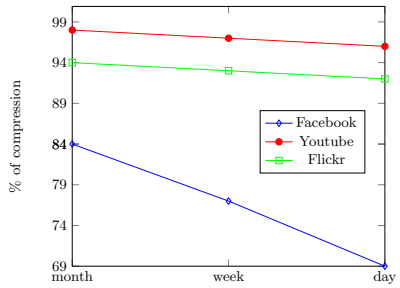

Figure 5: Compression ratio achieved by posting sharing

Posting Sharing. Finally, let us take a closer look at the posting sharing optimization by evaluating the reduction in the size of postings for various granularities as depicted in Figure 5. In general, we achieve compression ratios for the posting around 70% for FB, around 90% for FL and over 95% for YT. The compression ratio decreases with snapshot granularity due to the increase of the posting combinations. This is more evident for the FB dataset where the number of snapshots is higher.

6.2 Construction Time

In this set of experiments, we evaluate the time to construct the various indexes.

As seen in Figure 7, TRC is slower than TR, because of the additional time required for performing the bipartite matching. TRCH is even slower, since it also needs to construct the 2hop labels. We use the greedy algorithm for the bipartite matching and the INOUT strategy for computing the interval-2hop labels.

Constructing the TC for the whole graphs is prohibitive, since even for only 6,000 nodes, it takes over 22 hours, while

Table 6: Size (MB) per snapshot granularity

	Facebook			*YouTube*			*Flickr*		
	Days	Weeks	Months	Days	Weeks	Months	Days	Weeks	Months
VG	11	6	5	7.87	7.34	6.94	45.52	39.85	38.15
TR	0.58	0.47	0.42	44.28	21.28	14.98	141	73	41
TRC	0.22	0.08	0.07	3.21	1.92	1.46	2.89	2.27	1.88
TRCH	1,528	1,041	845	5,865	4,936	4,062	7,951	6,684	5,719

the TR construction takes just 0.72 seconds (Table 4).

Comparison of Different Bipartite Matching Algorithms. We also constructed the TRC using the Hungarian algorithm. For all datasets, the size of the resulting TRC is almost equal to the size of the TRC resulting from using the greedy algorithm (the difference is in the order of KB), thus confirming our expectation that greedy achieves a very close approximation of the optimal solution for social graphs. The Hungarian algorithm is much slower than greedy requiring an additional 1.5 hour for large datasets such as FL.

Comparison with 2hop for insert only. We adopted the pruned labeling algorithm proposed in [2] for distance queries to create an indexing scheme for historical reachability queries. Pruned labeling incrementally updates the index for each newly inserted edge, whereas in our approach we compute 2hop labels per snapshot. The pruned labeling algorithm does not support deletions, thus, we compare the two algorithms on the Facebook dataset without deletions. The pruned algorithm was found to be 5.4 times faster but it produced labels that were 12 times larger that the ones computed with our approach.

6.3 Query Processing

Let us now focus on query processing. In each experiment, we ran 500 historical reachability queries where the source and target nodes are chosen uniformly at random with the restriction that both nodes are present in the graph at the beginning and the end of the query interval. Queries involving nodes not present either at the beginning or the end of the query interval can be pruned fast by checking the lifespans of the nodes.

Online Traversal of the Version Graph. Let us first compare between an instant-based (INS) and an interval-based (INT) online traversal of the version graph for different time intervals (Figures 8 and 9). A general remark that holds independently of the method used to evaluate queries is that false conjunctive queries are faster than true conjunctive queries, since processing stops as soon as a time instant is found at which the two nodes are not reachable. Analogously, true disjunctive queries are faster than false disjunctive queries, since processing stops as soon as a time instant is found at which the two nodes are reachable.

Interval-based traversal is faster that instant-based traversal for almost all datasets and query types, since it can find the answer faster by searching for longer intervals. The only exception is FB and false conjunctive queries, where INS is slightly better. This happens because with INS, the search stops as soon as the first false answer is produced in any traversal. Hence, if this answer is found in the first few time instants of the query interval negative answers can be produced quickly for the smaller graph (i.e, the FB graph).

Online Traversal versus TimeReach. Let us now compare interval-based online traversal with query processing using the TR, TRC and TRCH approaches. The results for conjunctive queries are shown in Figure 10 and for disjunctive queries in Figure 11.

We see that all approaches are not significantly affected by the increase of the query interval due to fast posting lookups and short distances in the SCC graph for the TR and TRC, and the efficient implementation of edge lifespans for the version graph. We see that the TRC approach does not only produce a smaller structure than TR but it also attains faster query response for almost all datasets. TR is slower because for answering a query it needs to traverse the SCC graph per time instant when the query nodes do not belong to the same component. TRCH attains the fastest time when compared with all other approaches. The performance of TRCH is expected, since only two simple steps are needed: first to obtain the intersection $L_{in}(v) \cap L_{out}(u)$, and after that to check the lifespans $\mathcal{L}$ of the nodes in the intersection.

7. CONCLUSIONS

Most real-life graphs evolve over time. In this paper, we address the problem of efficiently answering historical reachability queries over such graphs. Such queries ask whether a node u was reachable from another node v during a time interval in the past. We have proposed an approach termed *TimeReach* that exploits the fact that most graphs consist of strongly connected components (SCCs). TimeReach maintains information about SCC membership for each node, and a graph which represents the links between the strongly connected components. We also maintain a condensed version graph which corresponds to the version graph of the SCCs evolution. Our extensive experiments with three real social network datasets show that TimeReach is storage-efficient and can be constructed incrementally with a small overhead. Historical queries are processed efficiently even when involving large time intervals.

There are many possible directions for future work. One such direction is exploiting TimeReach towards answering other types of historical queries, such as shortest path ones. Another direction concerns the distribution of TimeReach. Distribution may either be based on time or exploit the SCC evolution by placing together nodes that belong to the same SCCs.

8. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

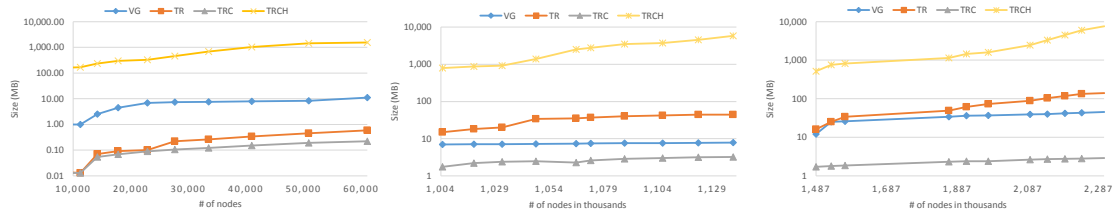

Figure 6: Size (log scale) for varying number of nodes in FB (left), YT (middle) and FL (right)

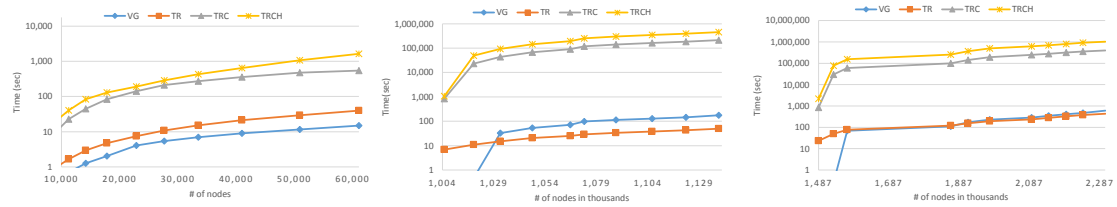

Figure 7: Construction time (log scale) for varying number of nodes in FB (left), YT (middle) and FL (right)

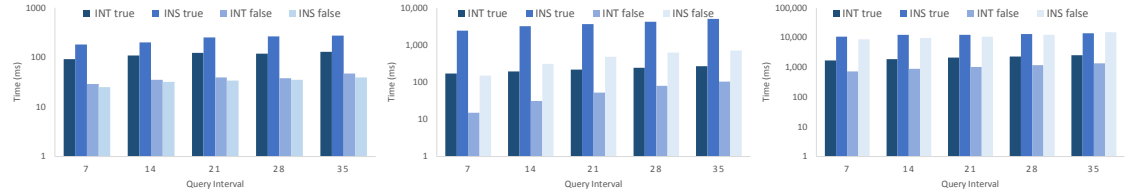

Figure 8: Query time (log scale) INS and INT for conjunctive queries in FB (left), YT (middle) and FL (right)

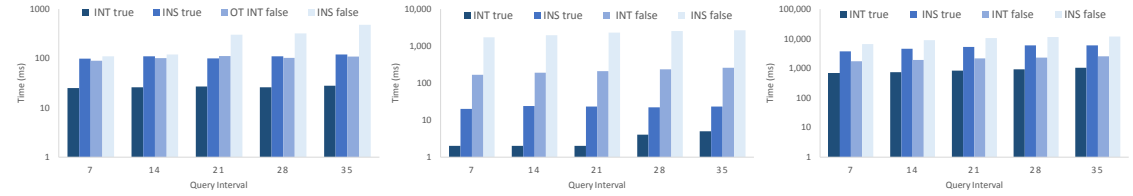

Figure 9: Query time (log scale) INS and INT for disjunctive queries in FB (left), YT (middle) and FL (right)

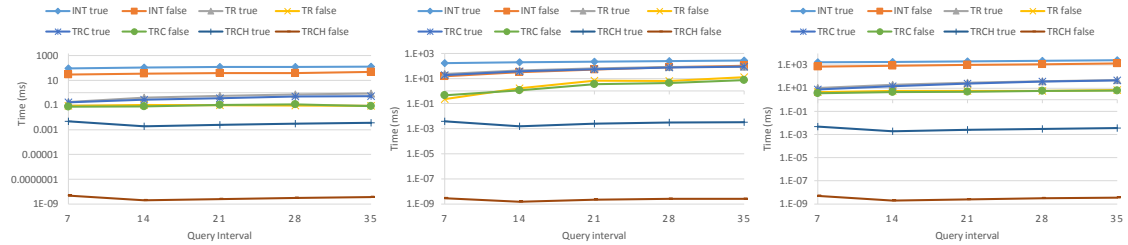

Figure 10: Query time (log scale) for conjunctive queries in FB (left), YT (middle) and FL (right)

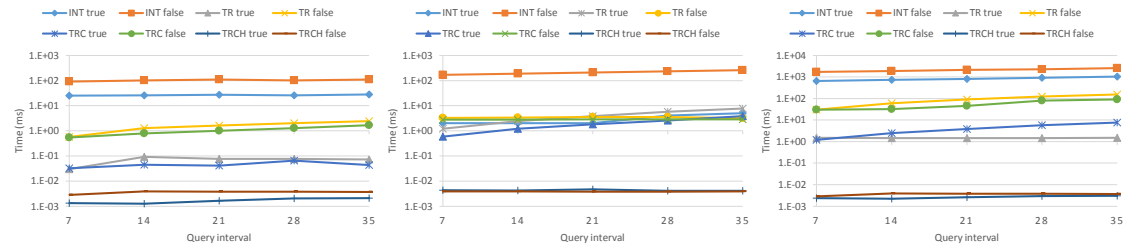

Figure 11: Query time (log scale) for disjunctive queries in FB (left), YT (middle) and FL (right)

9. REFERENCES

[1] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.

[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, pages 237–248, 2014.

[3] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW*, pages 845–854, 2008.

[4] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.

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

[6] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, pages 193–204, 2008.

[7] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[8] Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for approximate and exact maximum weight matching. *Arxiv preprint arXiv:1112.0790*, 2011.

[9] Wenyu Huo and Vassilis J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, page 38, 2014.

[10] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[11] Christian S. Jensen and Richard T. Snodgrass. Temporal element. In *Encyclopedia of Database Systems*, page 2966. 2009.

[12] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608, 2008.

[13] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, pages 997–1008, 2013.

[14] Georgia Koloniari, Dimitris Souravlias, and Evaggelia Pitoura. On graph deltas for historical queries. *WOSS*, 2012.

[15] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *ACM SIGKDD*, pages 611–617, 2006.

[16] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen Jr, Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The g* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, To appear, 2014.

[17] Alan G. Labouseur, Paul W. Olsen, and Jeong-Hyon Hwang. Scalable and robust management of dynamic graph data. In *VLDB*, pages 43–48, 2013.

[18] Alan Mislove. Online social networks: Measurement, analysis, and applications to distributed information systems. Rice University, Department of Computer Science, 2009.

[19] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, and Bobby Bhattacharjee Peter Druschel. Growth of the flickr social network. In *ACM SIGCOMM WOSN*, pages 25–30, 2008.

[20] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *ACM SIGCOMM IMC*, pages 29–42, 2007.

[21] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

[22] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, pages 237–255, 2004.

[23] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.

[24] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

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

[26] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD Conference*, pages 913–924, 2011.

[27] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *ACM SIGCOMM WOSN*, pages 37–42, 2009.

[28] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.

[29] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606, 2013.

[30] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. Grail: a scalable index for reachability queries in very large graphs. *VLDB J.*, 21(4):509–534, 2012.

[31] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *CoRR*, abs/1301.0977, 2013.