Time Traveling in Graphs using a Graph Database

Konstantinos Semertzidis
Computer Science & Engineering Department
University of Ioannina, Greece
ksemer@cs.uoi.gr

Evaggelia Pitoura
Computer Science & Engineering Department
University of Ioannina, Greece
pitoura@cs.uoi.gr

ABSTRACT

Most graph structured data, such as data created from the web, social, citation and computer networks, evolve over time. In this paper, we assume that we are given the history of a graph in the form of a sequence of graph snapshots. Our goal is to define the different types of queries that one can ask regarding the graph history and present an initial approach to storing graph snapshots and processing historical queries in a native graph database. We define three general types of historical queries, namely, historical graph queries, historical time queries and historical top-k queries. We present two representations of graph snapshots that use either a single or a multi-edge approach. We evaluate the two approaches experimentally for various types of historical reachability queries.

Categories and Subject Descriptors

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

General Terms

Measurement, Performance

Keywords

Graph Database, Historical Queries, Reachability

1. INTRODUCTION

In recent years, increasing amounts of graph structured data are made available from a variety of sources, such as social, citation, computer and hyperlink networks. Almost all such real-world networks evolve over time. In this paper, we focus on structural evolution, that includes node and edge additions and removals.

There has been a lot of interest on analytical processing and mining of evolving graphs, including among others developing models [14], discovering communities [2], and computing measures such as PageRank [3]. There has been also research on building graph engines tailored to supporting analytical processing in dynamic graphs, such as Kineograph [5] and Chronos [8]. Instead in this paper, we focus on online query-based processing of dynamic graphs.

We assume that we are given an evolving graph, in the form of a sequence of graph snapshots corresponding to the state of the graph at different time points. Our goal is to study the types of queries that one can ask on these sequence of snapshots. We call such queries *historical queries*. Although graph query processing has received a lot of interest, querying an evolving graph has received limited attention.

First, we provide a taxonomy of historical queries. We introduce three categories of historical queries, namely, historical graph queries, historical time queries and historical top-k queries. *Historical graph queries* are graph queries applied at past snapshots. For example, a historical graph reachability query may ask whether two nodes were reachable at some time interval in the past. *Historical time queries* are graph queries that ask about the time point or the time interval that a query had a specific result. For example, a historical time reachability query may ask for the time point at which two nodes become reachable for the first time. Finally, *historical top-k queries* ask for the nodes that had a property for the longest period of time. For example, a historical top-k reachability query may ask for the k pairs of nodes that remained connected for the longest interval.

We then address the problem of processing historical reachability queries in a native graph database. We study two approaches of storing graph snapshots, using either a single edge or multiple edges to represent connections that appear at different time points. We present algorithms for processing all different types of historical reachability queries using these approaches and experimentally compare their performance. Due to the native support of an edge-type traversal by the graph database, the multiple-edge approach outperforms the single-edge approach. Indexing boosts the performance of both approaches.

Previous work on historical queries is limited and includes only historical graph queries [1, 4, 9, 10, 11, 12, 15, 16] with the exception of [1] that studied one instance of a historical time query and the recent work in [17] that studies historical top-k graph patterns. Also, none of the previous approaches, with the exception of [4], is built on top of a native graph database. The work in [4] proposes a general approach for storing an evolving graph in a native graph database, specifically Neo4j, and experimentally tests a number of general graph queries that include past time instances. Our focus here is on structural updates and reachability queries. We

include an experimental comparison with their model.

The rest of the paper is structured as follows. In Section 2, we formally define historical reachability queries. In Section 3, we introduce two approaches to storing the evolution of graphs in graph databases and algorithms for processing historical reachability queries. In Section 4, we evaluate our approaches by presenting experimental results. In Section 5, we present related work. Section 6 concludes the paper.

2. HISTORICAL QUERIES

Most real world graphs evolve over time. In this paper, we focus on structural evolution, that includes node and edge additions and removals. We assume that time is discrete and use successive integers to denote successive points in time. Let $G = (V, E)$ be a 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 point t, that is, the set of nodes and edges that exist at time point 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.

One can think of two interpretations of time points. One interpretation is that of actual time, for example time point t may correspond to say February 9, 2015, 5:00am PDT. Another view is operational. In this case, a new time point is created, after a specific number of inserts and deletes of nodes and edges has occurred. We use the term *time granularity* to refer to how often a new time point 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.

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. $e \in E_{t_m}$).

Let us now discuss the type of queries that one can pose on evolving graphs. We call such queries *historical* to distinguish them from queries that consider only the current graph snapshot, G_{curr}. In the following, we first present various types of general historical queries and then formally define these types for the case of historical reachability queries.

Historical Graph Queries. The first category of historical queries include queries that are similar to current graph queries but refer to past snapshots. Let Q be any type of graph query, e.g., a reachability, shortest distance, or graph pattern query. The corresponding historical query Q_H on an evolving graph $\mathcal{G}_{[t_i,t_j]}$ is a pair (Q, I_Q), where I_Q is an interval $[t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$. Query Q_H is executed by applying query Q at all graph snapshots G_t, $t_l \leq t \leq t_m$ of $\mathcal{G}_{[t_i,t_j]}$, and returns as result an appropriately defined aggregation of these results. For example, let Q be a query that asks for the shortest path distance between nodes u and v and let $Q_H = (Q, [t_l, t_m])$. The shortest path distance between nodes u and v is computed at all graph snapshots G_{t_l}, G_{t_l+1}, ... G_{t_m} and these t_m - t_l + 1 distances are appropriately combined to produce the result of Q_H.

There are three general ways of combining the results: (a) use all snapshots, (b) use only one of the snapshots, and (c) an intermediate case, in which we use r of the involved snapshots. For example, in the case of shortest path queries, we may combine the results by returning in case (a) the average shortest path distance, in case (b) the minimum shortest path distance and in case (c) the minimum shortest path distance in at least r of the snapshots. Let us now define formally the three ways of combining results in the case of reachability queries.

Definition 2 (HISTORICAL REACHABILITY QUERY)**.**
Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a time interval $I_Q = [t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$ and a pair of nodes v, u:

(a) a *conjunctive historical reachability query* (CONJ) returns true, if there exists a path from u to v in all graph snapshots G_t, $t_l \leq t \leq t_m$ of $\mathcal{G}_{[t_i,t_j]}$.

(b) a *disjunctive historical reachability query* (DISJ) returns true, if there exists a path from u to v in at least one graph snapshot G_t, $t_l \leq t \leq t_m$, of $\mathcal{G}_{[t_i,t_j]}$,

(c) an *at least r historical reachability query* (LEAST) returns true, if there exists a path from u to v in at least r graph snapshots G_t, $t_l \leq t \leq t_m$, of $\mathcal{G}_{[t_i,t_j]}$.

In the special case in which $t_l = t_m$, we just apply Q on the single past snapshot G_{t_l} of $\mathcal{G}_{[t_i,t_j]}$. We call such queries *stab* (STAB) queries.

Historical Time Queries. Another type of queries pertinent to evolving graphs are queries that focus on the timing aspect. Such queries ask *when* an event happened. For example, depending on the type of query, we may ask when a specific graph pattern occurred, when two nodes become reachable, or when their shortest path distance was equal to a given value.

We make a distinction between queries that ask (a) when is the first time that an event happened, (b) what is the longest continuous interval that the event lasted, or (c) what is the total time that the event occurred. For reachability queries, we have the following types of historical time queries.

Definition 3 (HISTORICAL TIME REACHABILITY QUERY)**.**
Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a time interval $I_Q = [t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$ and a pair of nodes v, u:

(a) a *first time reachability query* (FIRST) returns the smallest time point t, $t_l \leq t \leq t_m$, such that, there exists a path from u to v in graph snapshot G_t and there is no path from u to v in any $G_{t'}$, $t_l \leq t' < t$,

(b) a *longest continuous time reachability query* (INTV) returns an interval $[t_{k_1}, t_{k_2}]$, $t_l \leq t_{k_1} \leq t_{k_2} \leq t_m$, such that, there exists a path from u to v in all graph snapshots G_t, $t_{k_1} \leq t \leq t_{k_2}$ of $\mathcal{G}_{[t_i,t_j]}$ and there is no longer interval that this holds,

(c) a *longest total time reachability query* (TOTAL) returns the time points t, $t_l \leq t \leq t_m$, such that there exists a path from u to v in G_t.

Historical Top-k Queries. The last type of historical queries are queries that ask for the top-k nodes that satisfy a condition for the longest time period. Depending on the query, this may mean what is the graph pattern that appears in the majority of graph snapshots, what are the pairs of nodes that remained reachable the longest, or what are the

nodes whose distance was below some given value for most of the time points. Again, we make a distinction between queries that ask for nodes that satisfy the property for the longest duration, either (a) continuously or (b) in total. For reachability queries, this gives us the following two types of top-k queries.

Definition 4 (HISTORICAL TOP-k REACHABILITY QUERY)**.** Given an evolving graph $\mathcal{G}_{[t_i,t_j]}$, a time interval $I_Q = [t_l, t_m]$, $t_i \leq t_l \leq t_m \leq t_j$ and a pair of nodes v, u and an integer k, $k > 0$:

(a) a *top-k continuous reachability query* (TOPK_I) returns a set of k pairs (u, v) of nodes u and v such that there exists a path from u to v in all graphs G_t in an interval of size d and there is no pair of nodes u', v', for which a path from u' to v' exists in all graphs in an interval of size larger than d,

(b) a *top-k total reachability query* (TOPK_T) returns the k pairs (u, v) of nodes u and v such that there exists a path from u to v in the largest number of graph snapshots G_t, $t_l \leq t \leq t_m$.

3. HISTORICAL QUERIES ON A GRAPH DATABASE

In this section, we first present two different approaches of representing an evolving graph within a graph database and then algorithms for processing historical reachability queries using these representations.

As a running example, we use a bibliographical database where nodes correspond to *AUTHORS* and there is a *PUBLISH* edge between two AUTHORS if they are co-authors. Time granularity corresponds to years.

3.1 Single and Multi Edge Representation

In the *single-edge approach*, the lifespan of a node or edge is modeled using a label (i.e., an attribute or property) of the corresponding node or edge. Figure 1 shows an example of edge lifespans represented by a single label of type String. For example, the co-authorship between authors A, C in 2010, and 2012, is represented by the edge with label "2010,2012". Graph evolution can be tracked by processing the node and edge lifespan labels. Thus, to obtain the graph snapshot G_t for a time point t, we get all edges and nodes and then keep only those edges and nodes whose lifespan label contains t.

While in the single-edge approach there is at most one edge between two nodes, the *multi-edge approach* uses a different edge type between two nodes for each time point of the lifespan of the edge. For instance, in order to represent co-authorship between authors A, B, C in 2010, 2012, and 2014, we use three different labeled types of edges to connect A, B, and C. An example is shown in Figure 2 which depicts co-authorship between A, B and C in 2010, 2012, and 2014 using multiple type of edges. Multiple type of edges provide an efficient way of retrieving graph snapshots, since a graph database supports an efficient traversal of edges of a given type. Note that when a node u is connected by an edge of a type that corresponds to a time point t, it is obvious that u existed in t. Thus to get information about the lifespan of u, we access its edges, which is a fast process, since graph databases provide efficient indexing for this process.

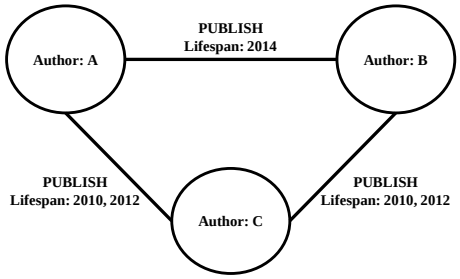

Figure 1: Time as attribute/label

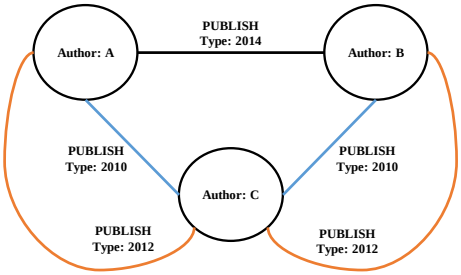

Figure 2: Time as a different edge type

For both the single-edge and the multi-edge approaches, we explore whether an index on the lifetime property of each node improves the performance of retrieving a snapshot of the graph. We build an index within the graph database (shown in Figure 3) by creating a new node type TIME where each node of the given type has a unique value that corresponds to a specific time point. A TIME node that denotes a time point t has relations to all AUTHOR nodes that existed at time t. To obtain all nodes that exist in an interval, we get the neighbors of the TIME nodes that correspond to this interval.

3.2 Historical Reachability Query Processing

All graph databases provide a $BFStraversal$ method that visits all edges of a given type. This method starts from a source node, visits all its neighbors at distance 1 by traversing edges of the given type, then all its neighbors at distance 2, and so on.

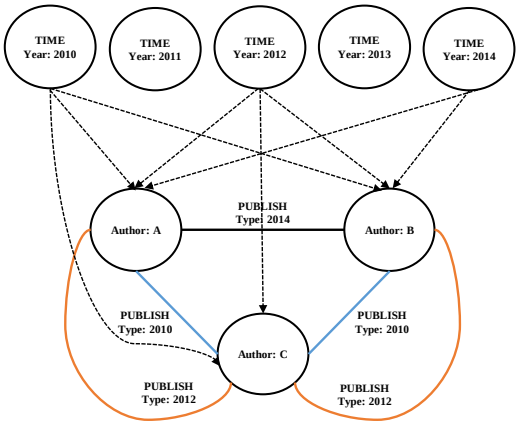

Figure 3: Time index for year 2010 to 2014

For answering if two nodes are reachable in the multi-edge approach, we use this method. For example, to find if two nodes are connected in a given interval I, we call the $BFStraversal$ method for each time instant t_i in I passing as an argument the edge type that corresponds to t_i.

However, we cannot use the $BFStraversal$ method for the single-edge approach, since we need to post-process the lifespan label of each edge to find the time instants where the nodes were reachable. The $BFStraversal$ method provided by the deployed graph database does not return the path visited but just the reachable nodes. Thus, we implemented our own BFStraversal algorithm which processes the edge lifespans.

The time index can be used similarly in both single-edge and multi-edge approaches to prune some computations. For example, for the LEAST query that asks whether nodes u and v are reachable at least r time points in I, we can first check using the index whether both nodes were active at the same time points at least r times. If they were not active, we do not need to traverse the graph.

To summarize, (a) in the single-edge approach, we use our own implementation of the $BFStraversal$ method and (b) in the multiple-edge approach, we invoke the provided $BFStraversal$ method once for each edge type corresponding to the time points in the query interval.

Let us now discuss in more detail how to process the different types of historical reachability queries that ask for a path from u to v in interval I. A CONJ query returns true, when there is no t_i in I where v is not reachable from u. A DISJ query returns true, when the first t_i in I is found in which v is reachable. Finally, for a LEAST query, we keep a counter c of the time points in I that v is reachable. If the counter reaches the given r, the query returns a positive answer, otherwise it stops when the sum of the counter and the remaining time instants is less than r.

Historical time reachability queries return time points or time intervals. For a FIRST query, we return the first t_i when v is reachable. For a INTV query, we keep a counter c for the consecutive times that v is reachable and a max variable of the current maximum c. For each t_i, that v is not reachable, c is reset and max is updated if $c > max$. The query stops and returns the interval corresponding to the max value, when c and the remaining time instants are less than max. Finally, for a TOTAL query, we return all time points in I where v is reachable.

For the top-k historical reachability queries, we use the time index to obtain the top-k (active) nodes u_{top}, that is the k nodes that exist for the longest period in I. In particular, for each node u we found in a time instant of I we increase a counter that denotes the number of instants that u is active. We also use a Min Heap structure to keep the top-k pairs of nodes that were connected for the longest interval (TOPK_I query) or for the largest number of time points (TOPK_T query). The Min Heap stores each pair as a triple $(u, v, value)$ where $value$ is a counter that keeps the longest interval or the largest number of times that u and v were connected.

For a TOPK_I query, we start traversing from each top active node u_{top} for each time instant in I. Intuitively, top active nodes have more active paths to other nodes. For each node v reachable from u_{top}, we increase a counter $C(v)$, each time v is reachable. We also keep a $max(v)$ variable for the maximum $C(v)$. Each time v is not a reachable from u_{top}, $C(v)$ is reset and $max(v)$ is updated if $C(v) > max(v)$. In the end of the traversal from u_{top}, we insert in the Min Heap the triple $(u_{top}, v, max(v))$ if $max(v)$ is larger than the smallest element of the Min Heap. The query stops when the Min Heap has size k and its minimum element is larger or equal to the lifetime in I of the remaining top active nodes. Processing of a TOPK_T query is similar. The only difference is that in the Min Heap, we store the number of time points instead of the duration of the interval.

4. EVALUATION

As our graph database, we use Sparksee [6] that supports fast loading of the graph data and efficient operations that scan all edges in the graph. Sparksee is based on a compact representation that uses bitmaps and highly compressible data structures [7]. As our dataset, we used the whole DBLP dataset [1] for the interval [1958, 2015]. Each graph snapshot corresponds to a year in this interval.

We ran all queries on a system with a quad-core Intel Core i7-4770 3.4 GHz processor and 32 GB memory. We only use one core for all experiments.

Storage. We created two different graph database instances (GDBs). The first graph database instance stores our dataset following the single-edge approach, whereas the second follows the multi-edge approach.

Table 1 shows the characteristics of each graph database instance. Single-edge differs from multi-edge in the number of edge types, since the second one uses a different edge type for each time point, which leads to a larger size. The index nodes size states the number of time points which in our case is 58 years. The edge types for single-edge are two, one type represents the PUBLISH edge and the other one the index edge. Multi-edge has 59 types, one type for the index edge and 58 types for each year in [1958, 2015].

Historical Query Processing. To evaluate the performance of both true and false queries, we generated for each query type 250 true and 250 false queries.

For CONJ, DISJ, LEAST, FIRST, TOPK_I, and TOPK_T, the query interval is $I = [2005, 2014]$, for STAB, the time point is a random year within I, for INTV, TOPK_I the interval is $I = [1958, 2015]$. For LEAST, r was randomly chosen from [2, 9], while in TOPK_I, and TOPK_T, k is equal to 10.

Table 2 reports the average time of true and false queries, on both graph instances. Also, we report the query time when the time index is used, as well the average execution time of TOPK_I, and TOPK_T on the multi-edge GDB instance.

Queries that ask for events that have the longest duration, either continuously or in total require the most time to be processed, (since we need to check long time intervals) followed by queries that seek for the largest number of time points that something holds or the longest continuous interval that something holds.

Comparing the GDB instances, we notice that queries are faster when using multiple types of edges to represent the time points. This can be explained by the fact that using a single edge type requires the processing of the edge to find if a time point is contained in the lifespan label.

A general remark is that false conjunctive queries are faster than true conjunctive queries, since processing stops as soon as a time point is found at which the two nodes

[1] http://dblp.uni-trier.de/

Table 1: Graph database properties

# GDB	# Nodes	# Edges	# Index Nodes	# Index Edges	# Edge Types	Size (MB)
Single-edge	1,013,762	3,849,319	58	2,542,405	2	538
Multi-edge	1,013,762	5,186,596	58	2,542,405	59	660

Table 2: Queries average time (ms)

	Single-edge GDB				*Multi-edge GDB*			
	With Index		*Without Index*		*With Index*		*Without Index*	
	true	false	true	false	true	false	true	false
STAB	1.23	1,162	3.23	4,955	0.43	8	0.18	45
CONJ	5,790	0.24	9,481	6,464	433	0.25	424	21
DISJ	269	633,413	268	543,881	9.52	227	9.32	492
LEAST	54,038	13,777	55,454	285,467	113	18	114	397
FIRST	68,764	42,476	251,728	375,220	65	61	178	457
INTV	763,283	434,196	763,828	440,672	827	72	1,619	570
TOTAL	1,352,035	542,213	1,364,020	632,630	966	65.56	1,691	500
TOPK_I	13,020 *(Multi-edge with Index)*							
TOPK_T	12,650 *(Multi-edge with index)*							

Table 3: Top 10 pairs of authors from TOPK_I and TOPK_T on Multi-edge GDB and $I = [2005, 2014]$

	Pairs
1	Ravishankar K. Iyer – Zbigniew Kalbarczyk
2	Wesley De Neve – Rik Van de Walle
3	M. Brian Blake – Walter Binder
4	Juan A. Rodriguez Aguilar – Axel Polleres
5	S. V. N. Vishwanathan – Zbigniew Kalbarczyk
6	Hans-Peter Kriegel – Fabio Gadducci
7	Kenneth R. Koedinger – Jie Xu
8	Bernhard Steffen – Frank Seide
9	Stefania Gnesi – Maurice H. ter Beek
10	Mariangiola Dezani-Ciancaglini – Luca Padovani

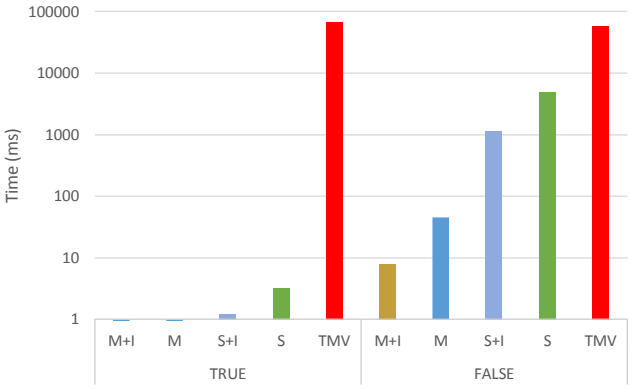

Figure 4: STAB time (log scale) on different approaches

are not reachable. Analogously, true disjunctive queries are faster than false disjunctive queries, since processing stops as soon as a time point is found at which the two nodes are reachable. Also an observation that holds independently of the graph GDB used to evaluate queries is that the time index boosts query processing. We gain more speed in false queries than true ones, since we can prune traversals from nodes that are not active in a given time point or in the whole interval. For example, if we seek to find the longest interval during which A and B were connected and there is not any time point that both authors were active, then a false answer is returned without executing any traversal.

Table 3 shows the top 10 authors pairs returned from TOPK_I, and TOPK_T. Both queries return the same pairs because the authors of each pair were reachable in the whole interval (10 years). Thus, the authors that were connected for the longest interval are the same with the authors that are connected the most time in the past. We clarify that the top-k processing steps that were followed in TOPK_I, and TOPK_T stop when they find the first k pairs that meet the requirement. Hence, ties, i.e., pairs that have the same property may not be reported.

Comparison with the Time-Varying Approach. Finally, we implemented the data model introduced in [4] and tested its performance for the STAB query. The approach in [4] introduces a specific node to model the interaction between two nodes at a specific time point. They also use a hierarchical index to support different time granularities, which is an issue that we do not address here, thus, we do not implement such an index.

We created a new type of node PAPER which denotes the interaction of publishing a paper and an AUTHOR node type for the authors. We connect with each PAPER its authors using an edge type PUBLISH. For the time index, we connect each AUTHOR and PAPER node to the time index nodes to which they belong. For example, if authors A and B wrote a paper P together in t then from the time index node that corresponds to t, we create edges that connect the time index node with the A, B and P nodes. To find if two authors A and B are reachable, we have to obtain the authors from each PAPER node that A is connected and from them to obtain their co-authors. We repeat this process until we find B. This process is costly for finding PAPER nodes that were active at a specific time point, since we check the time index for each PAPER node to see if it was active in that time.

Running a STAB query using this approach requires 67.8

seconds for true queries and 58.5 seconds for false queries (shown in Figure 4), while our best approach requires only 0.43 and 8 milliseconds for answering true and false queries respectively. This can be explained by the fact that their model has not been designed for answering historical reachability queries but for querying the presence of objects in a number of given time points.

5. 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 query processing in evolving graphs has been on efficiently storing and retrieving graph snapshots. In this paper, our focus is on defining different types of historical queries and on processing them in a graph database.

Historical query processing in these approaches requires as a first step reconstructing the relevant snapshots. Then, queries are processed through an online traversal on each of the snapshots. Various optimizations for reducing the storage and snapshot reconstruction overheads have been proposed. Optimizations include the reduction of the number of snapshots that need to be reconstructed by minimizing the number of deltas applied [11], using a hierarchical index of deltas and a memory pool [10], avoiding the reconstruction of all snapshots [15], and improving performance by parallel query execution and proper snapshot placement and distribution [13].

Recent work also addresses indexing for historical reachability queries through an index that contains information about strongly connected components membership at various time points [16], indexing for historical shortest path distance queries [1, 9] and indexing for durable pattern queries on historical graphs [17].

Finally, the only work on evolving graphs using a native graph database we are aware of is the work in [4] where the authors use Neo4j. Their evolving graph besides structural updates, also includes time-varying attributes. They do not provide a taxonomy of historical graph queries, instead they implement a number of historical queries that also include attributes.

6. CONCLUSIONS

In this paper, we studied queries on evolving graphs. We have proposed a taxonomy of historical graph queries, that are queries that involve past graph snapshots. We presented different approaches of storing and retrieving an evolving graph in a graph database by either using a single-edge with a lifespan attribute or multiple-edge types where each type corresponds to a different time point. We have proposed algorithms for evaluating historical reachability queries and evaluated our approaches using the DBLP dataset.

There are many possible directions for future work. One such direction is exploiting our algorithms towards answering other types of historical queries, such as shortest path ones. Another direction concerns historical queries that take into account attributes on the nodes and edges as well evolving graphs with time-varying attributes.

7. REFERENCES

[1] 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.

[2] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[3] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *PVLDB*, 4(3):173–184, 2010.

[4] Ciro Cattuto, Marco Quaggiotto, André Panisson, and Alex Averbuch. Time-varying social networks in a graph database: a neo4j use case. In *GRADES*, page 11, 2013.

[5] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[6] Sparksee Graph Database. http://www.sparsity-technologies.com/.

[7] David Dominguez-Sal, P. Urbón-Bayes, Aleix Giménez-Vañó, Sergio Gómez-Villamor, Norbert Martínez-Bazan, and Josep-Lluis Larriba-Pey. Survey of graph database performance on the HPC scalable graph analysis benchmark. In *WAIM*, pages 37–48, 2010.

[8] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *EuroSys*, page 1, 2014.

[9] Wenyu Huo and Vassilis J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, page 38, 2014.

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

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

[12] 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*, 2014.

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

[14] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

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

[16] Konstantinos Semertzidis, Kostas Lillis, and Evaggelia Pitoura. Timereach: Historical reachability queries on evolving graphs. In *EDBT*, pages 121–132, 2015.

[17] Konstantinos Semertzidis and Evaggelia Pitoura. Durable graph pattern queries on historical graphs. In *ICDE*, 2016 (to appear).