Historical Traversals in Native Graph Databases

Konstantinos Semertzidis(✉) and Evaggelia Pitoura

Department of Computer Science and Engineering,
University of Ioannina, Ioannina, Greece
{ksemer,pitoura}@cs.uoi.gr

Abstract. Since most graph data, such as data from social, citation and computer networks evolve over time, it is useful to be able to query their history. In this paper, we focus on supporting traversals of such graphs using a native graph database. We assume that we are given the history of an evolving graph as a sequence of graph snapshots representing the state of the graph at different time instances. We introduce models for storing such snapshots in the graph database and we propose algorithms for supporting various types of historical reachability and shortest path queries. Finally, we experimentally evaluate and compare the various models and algorithms using both real and synthetic datasets.

Keywords: Graph database · Historical traversals · Reachability · Path computation

1 Introduction

Recently, increasing amounts of graph structured data are made available from a variety of sources, such as social, citation, computer, hyperlink and biological networks. Almost all such real-world networks evolve over time. Querying the evolution of such graphs is an important and challenging problem.

In this paper, we assume that we are given the history of an evolving graph in the form of a sequence of graph snapshots representing the state of the graph at different time instances. Our focus is on efficiently storing and querying these snapshots using a native graph database. Native graph databases offer an attractive means for storing and processing big graph datasets.

To store the sequence of graph snapshots in a graph database, we propose models based on associating with each node and edge, its lifespan, i.e., the time intervals, during which the node and edge is valid. The multi-edge approach (ME) uses a different edge type for each of the time instances during which the edge was valid. The single-edge approaches use a single edge annotated with a complex type for representing the lifespan of the edge. We consider two single-edge approaches, one that models the lifespan as an ordered list of time instances (SETP), and one that uses an interval representation (SETI).

We also introduce historical graph traversals that consider paths that existed in a sufficient number of graph snapshots. We exploit variants of two types of historical traversals, reachability and shortest paths. Historical reachability

M. Kirikova et al. (Eds.): ADBIS 2017, LNCS 10509, pp. 167–181, 2017.
DOI: 10.1007/978-3-319-66917-5_12

queries ask whether two nodes are connected in some time instance, in all time instances, or in a sufficient number of time instances. Historical shortest path queries ask for the shortest path between two nodes posing requirements on the lifespan of such paths. We present algorithms for processing historical queries for both the multi-edge and the single-edge approaches.

We have implemented our approach in the Neo4j graph database and present experimental results using both real and synthetic datasets. For very short-lived edges, using multiple edges to represent lifespans, seems to work well by taking advantage of the built-in traversal methods of the native graph database. However, for all other cases, using the interval-based approach to represent lifespans (SETI) proves more efficient both in terms of processing time and storage. We also present a case study regarding connectivity among authors of different conferences through time.

Related Work: There has been recent interest on analytical processing and mining of evolving graphs, including among others developing models [15], 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 [7]. However, our focus here is on query processing.

There has been some work on historical query processing. The common assumption is that the graph is either kept in main memory or is stored in disk, but not in a native graph database. Most research assumes as a first step the reconstruction of 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 [12], using a hierarchical index of deltas and a memory pool [11], avoiding the reconstruction of all snapshots [17], and improving performance by parallel query execution and proper snapshot placement and distribution [14]. Other research considers in-memory processing of specific types of historical queries [1,4,9,13,18,19].

Very few works [4,6,8,20] are built on top of a native graph database. In particular [4] proposes an approach for storing time-varying networks in the Neo4j graph database using a hierarchical time index to support snapshots with different granularity (e.g., months and days). They do not discuss historical traversal queries, but, instead consider retrieving specific snapshots. In [6] the authors focus on graph data with structural changes, and present time logs that capture when an event has occurred (i.e. add/remove of edge/node) in the history of the graph. Although, their indexes are used to retrieve fast a state of the graph in a given period they are not designed for supporting historical traversal queries. A short discussion of the storage models ME and SETP is made in position paper [20]. Finally, the work in [8] targets specific types of graphs with static structure but frequent changes in node and edges properties. Our focus here is on structural updates and reachability and path queries.

Paper outline: The rest of this paper is structured as follows. In Sect. 2, we formally define historical traversal queries. We introduce three approaches for storing the graph snapshots in graph databases in Sect. 3 and algorithms for processing historical traversal queries in Sect. 4. In Sect. 5, we experimentally evaluate the different approaches. Section 6 concludes the paper.

2 Traversals in Historical Graphs

In this section, we first define historical graphs and then introduce traversal queries on them.

2.1 Historical Graphs

Graphs are used to represent relationships between entities where entities are modeled as nodes and relationships as edges between them. Labels may be assigned to both edges to capture different types of relationships and to nodes to capture node attributes. Formally, a node and edge labeled graph G is a tuple $(V, E, \lambda^V, \lambda^E)$ where V is a set of nodes, $E \subseteq V \times V$ is a set of edges, $\lambda^V : V \rightarrow L^V$ and $\lambda^E : E \rightarrow L^E$ are labeling functions that map a node and an edge to a label from a set L^V of node labels and a set L^E of edge labels respectively. Some graph databases also support an extension of labeled graphs, termed *property graphs*, where instead of labels, a set of property-value pairs is associated with nodes and edges. Such pairs are also sometimes called *attributes*.

Most real world graphs evolve over time. New nodes and edges are added, and existing nodes or edges are deleted. We assume that time is discrete and use successive integers to denote successive points in time. We use $G_t = (V_t, E_t, \lambda_t^V, \lambda_t^E)$ to denote the *graph snapshot* at time instance t, that is the sets of node, edge and labeling functions that exist at time instance t.

Definition 1 (Historical graph)**.**
A historical 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 Fig. 1 which depicts a historical graph $\mathcal{G}_{[1,5]}$ consisting of five graph snapshots $\{G_1, G_2, G_3, G_4, G_5\}$. Nodes and edge labels are omitted for simplicity. Note that the granularity of creating time instances and graph snapshots may vary. For example, we may create a new graph snapshot at every second, hour or day.

We use the term *lifespan* (ls) to refer to the period of time that a graph element, that is, a node, edge, or labeling function, existed. Lifespans are sets of time intervals. Set of time intervals are also known as *temporal elements* [10]. For example, the lifespan $ls((u_1, u_3))$ of edge (u_1, u_3) in Fig. 1 is equal to $\{[1, 1], [3, 4]\}$ meaning that edge (u_1, u_3) existed in graph snapshots G_1, G_3 and G_4. We also define a useful operation on interval sets, called *time join* [18]. Given two sets of intervals $\mathcal{I}$ and $\mathcal{I}'$, their time join $\mathcal{I} \otimes \mathcal{I}'$ is the set of time intervals that includes the time instances that exist in both $\mathcal{I}$ and $\mathcal{I}'$. For example, $\{[1, 3], [5, 10], [12, 13]\} \otimes \{[2, 7], [11, 15]\}$ is equal to $\{[2, 3], [5, 7], [12, 13]\}$.

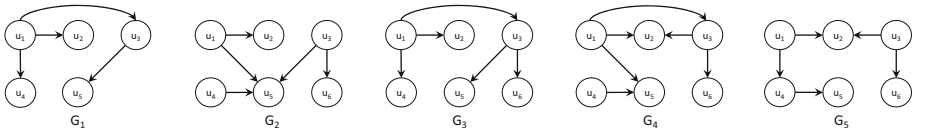


Fig. 1. Example of a historical graph

2.2 Historical Traversal Queries

A graph traversal allows the navigation of the structure of the graph and is a fundamental graph query. In an abstract form, a traversal query Q can be expressed as a path query $Q = u \xrightarrow{\alpha} v$, where α specifies conditions on the paths that we wish to traverse and u, v denote the starting and ending points of these paths. The starting and ending points can be specific nodes or properties of the nodes, or a mix of both. The expression α involves constraints on the properties (or, labels) of the nodes and edges in the path. For example, we may look for paths connecting two people in a social network with edges labeled as "friends".

Traversals retain the paths from u to v that satisfy α. In general, there are may be many such paths, even an infinite number, if there are cycles in the dataset. Thus, besides maintaining all possible paths, various other semantics may be associated with the evaluation of traversals. Common ones are retaining only the shortest paths, or only paths that consist of no-repeated nodes or edges.

We now define traversal queries on historical graphs. First, let us define the lifespan of a path. Let $\mathcal{G}_{[t_i,t_j]}$ be a historical graph and $p = u_1 u_2 \dots u_m$ be a path of m nodes where $u_k \in \cup_{t_l=t_i}^{t_j} V_{t_l}$, $1 \leq k \leq m$. We define the lifespan, $ls(p)$, of path p as follows: $ls(p) = ls((u_1, u_2)) \otimes ls((u_2, u_3)) \dots \otimes ls((u_{m-1}, u_m))$. For example, the lifespan of path $u_1 u_3 u_6$ of $\mathcal{G}_{[1,5]}$ in Fig. 1 is $\{[3, 4]\}$.

Definition 2 (Historical Traversal Query)**.**
A traversal query Q_H on a historical graph, $\mathcal{G}_{[t_i,t_j]}$, called a historical traversal query, is a tuple (Q, $\mathcal{I}$, L) where Q is a traversal query $Q = u \xrightarrow{\alpha} v$, $\mathcal{I}$ is a set of time intervals and L is a positive integer. For a path p, let $D(p) = ls(p) \otimes \mathcal{I} \otimes [t_i, t_j]$. Q_H retains the paths p from u to v in $\mathcal{G}_{[t_i,t_j]}$ that satisfy α and for which in addition $D(p)$ contains at least L time instances.

Intuitively, we ask that the paths retained by a historical query exist in at least L of the graph snapshots. At one extreme a path must appear at least once in the graph history, in which case $L = 1$. At the other extreme, a path must appear in all time instances, in which case L must be equal with the number of time instances in $\mathcal{I} \otimes [t_i, t_j]$.

A traversal query may produce different outputs. For example, the output may be the set of the retained paths, or, the set of nodes, or edges on the retained paths. Furthermore, in the special case of reachability queries, the output of the traversal is boolean, i.e., true if there exists a path, and false otherwise. Without loss of generality, in this paper we focus on reachability shortest path queries. Additional semantics may be associated with the output of historical traversals.

For example, for reachability queries, we may ask that two nodes are reachable in at least one time instance (*disjunctive queries*), in all time instances (*conjunctive queries*), or in at least-k time instances. For shortest path queries, we may ask, for example, for the earliest shortest path (ESP), for the shortest among the paths that existed in all time instances (stable shortest path (SSP)), or, for the shortest among the paths that existed in at least-k snapshots (KSP).

3 Storing Historical Graphs

In this section, we present different approaches for representing a historical graph in a native graph database. The basic idea is to augment each graph element with its lifespan. For edges and nodes, lifespans are stored as labels (i.e., property, attribute) of the corresponding edge and node. Based on the type of labels used, we have two different approaches.

Multi-edge Representation. The *multi-edge approach* (ME) utilizes a different edge type between two nodes u and v for each time instance of the lifespan of the edge (u, v). The multi-edge representation of the historical graph $\mathcal{G}_{[1,5]}$ of Fig. 1 is depicted in Fig. 2. For instance, to represent a relationship between nodes u_1, u_3 with lifespan $\{[1, 1], [3, 4]\}$, we use three edges with different labels to connect u_1 and u_3. Since all native graph databases provide efficient traversal of edges having a specific label, the ME approach provides an efficient way of retrieving the graph snapshot G_t corresponding to time instance t. Similarly, multiple labels are associated with each node.

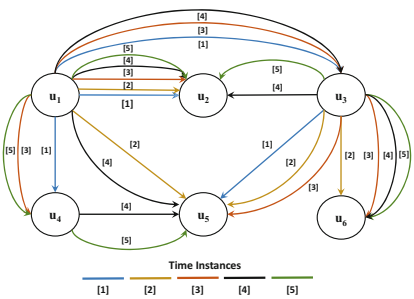
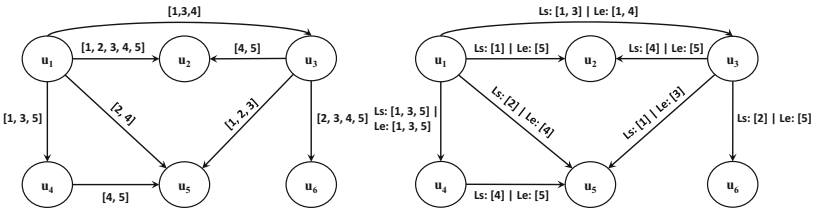


Fig. 2. ME representation of the historical graph of Fig. 1 (nodes labels are not shown for clarity)

Single-edge Representation. The *single-edge approach* uses a single edge between any two nodes appropriately labeled with the lifespan of the edge. To represent the lifespan of an edge or node, we consider two different approaches. In the *single-edge with time points approach* (SETP), the lifespan of a node or edge is modeled using a label that is a sorted list of the time instances in their lifespan. The SETP representation of the historical graph $\mathcal{G}_{[1,5]}$ of Fig. 1 is shown in Fig. 3(a).

For example, the lifespan of edge (u_1, u_3) is now represented by a single edge having as label [1, 3, 4]. In the *single-edge with time intervals approach* (SETI), we use Ls and Le, each one an ordered list of m elements, where m is the number of time intervals in the lifespan of the edge or node. In particular, $Ls[i]$, $1 \leq i \leq m$, denotes the start of the i-th interval in the lifespan, while $Ls[i]$ the end of the interval. An example is shown in Fig. 3(b). With the single-edge approaches, retrieving the graph snapshot G_t at time instance t requires further processing of the related labels.



(a) SETP representation (b) SETI representation

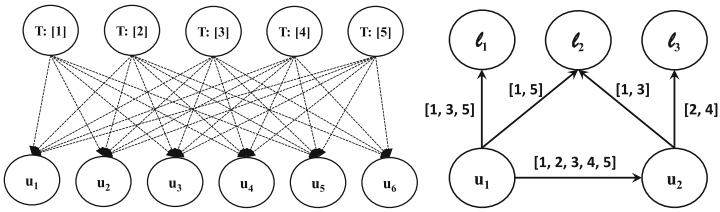
Fig. 3. Single-edge representations of the historical graph of Fig. 1 (nodes labels are not shown for clarity)

Indexing. For faster retrieval of specific graph snapshots, we build an index within the graph database by creating a new node type T where each node of the given type has a unique value that corresponds to a specific time instance. A T node that denotes a time instance t is connected with all nodes that existed at time instance t. To retrieve the nodes that exist in a time interval, we get the neighbors of the T nodes that correspond to this interval. Figure 4(a) shows the index of the historical graph in Fig. 1.

Time-varying labels. Finally, we discuss how to store labels that change over time. Current graph databases do not support versioning on labels and thus we need to create for each unique label value l, a new node of type l. We connect all nodes or edges that have value l at some time instance with the node representing l using one of the three edge approaches presented previously. Doing so, we only store each label once and to retrieve the labels of a node u in a time interval, we retrieve all the nodes type of l that are connected to u by edges that refer to the time instances in the interval. In Fig. 4(b), we depict an example of storing the time-varying labels of two nodes u_1, u_2 using SETP.

4 Processing Historical Traversal Queries

In this section, we focus on processing historical traversal queries in native graph databases. For simplicity, we consider a single interval I, but the algorithms easily extend to sets of time intervals.



(a) Time index (b) Time-varying labels

Fig. 4. (a) Time index of the historical graph of Fig. 1 and (b) an example of time-varying labels

Multi-edge Representation. A basic functionality provided by all native graph databases is a TRAVERSALBFS method that implements a BFS traversal of all edges of a specific type (i.e., with a specific label) starting from a source node. At each step, TRAVERSALBFS returns either the current traversed node or all the previously traversed nodes in a form of a path. One approach for retrieving the paths that exist between two nodes u and v during a time interval I is to invoke TRAVERSALBFS starting from u once for each time instance t in I and then combine these results. Another approach is to process paths an edge-at-a-time. Starting from u for each time instance t in I we traverse only the edges of type t until we reach v. Which of the two approaches is more efficient depends on the type of the traversal query under consideration.

For reachability queries where we ask that two nodes are reachable, without posing any requirement on the lifespan of the paths that connect them, the approach that uses the built-in TRAVERSALBFS is more efficient. We invoke TRAVERSALBFS for each time instance t in I until: (a) for disjunctive queries: the first time instance that we find v, (b) for conjunctive queries: the first time instance that we do not find v and (c) for the least-k queries: when we find v in at least k time instances, or the remaining time instances are not enough to get reachability at k time instances.

For queries that require that the paths exist in at least $L > 1$ time instances, using the TRAVERSALBFS method is in general expensive, since we retrieve all paths at each time instance, even those paths that appear only in a single time instance. Thus, TRAVERSALBFS is used only for the earliest shortest path (ESP) queries, where it returns the shortest path that connects u to v in the first time instance. For stable (SSP) and at least-k (KSP) shortest path queries, we use the second approach. We traverse the edge type that refers to the first time instance in I and we continue the traversal only if for each edge (w, x) there are all (SSP) or at least k (KSP) type of edges (w, x) that refer to other time instances in I.

Single-edge Representation. For the *single-edge approaches*, we cannot use the TRAVERSALBFS, since we need to post-process the lifespan label of each edge to determine the time instances where the edges were active. Thus, we implemented our own TRAVERSALBFS algorithm which traverses edges that are

Algorithm 1. (SETP-SETI) Conjunctive-BFS(u, v, I)

Require: nodes u, v, interval I
Ensure: True if v is reachable from u in all time instances in I and false otherwise

```
1: create a queue N, create a queue INT
2: enqueue u onto N, enqueue I onto INT
3: while N ≠ ∅ do
4:     n ← N.dequeue()
5:     i ← INT.dequeue()
6:     for each e ∈ n.getEdges() do
7:         I_e ← TIME_JOIN(e, i)
8:         if I_e = ∅ then
9:             continue
10:        end if
11:        w ← r.getOtherNode(n)
12:        if w = v then
13:            R ← R ∪ I_e
14:            if R ⊒ I then
15:                return true
16:            end if
17:            continue
18:        end if
19:        if IN(w) ⋣ I_e then
20:            IN(w) ← IN(w) ∪ I_e
21:            enqueue w onto N
22:            enqueue I_e onto INT
23:        end if
24:    end for
25: end while
26: return false
```

alive in the given interval. We present in Algorithm 1, the algorithm for processing conjunctive reachability queries. Algorithm 1 can be used for processing all other types of historical queries with only small modifications.

Since a node v may be reachable from u through different paths at different graph snapshots, we maintain an interval set R with the part of $ls(u \rightarrow v) \cup I_e$ covered so far (line 13), where I_e is the intersection of the lifespan of an edge with a given interval. The traversal ends when R covers the whole query time interval I (lines 14–16).

To retrieve I_e, we use method TIME_JOIN (line 7) and getOtherNode(n) which given a node n that is attached to an edge, returns the other node (line 11). In SETP, TIME_JOIN retrieves the lifespan label from the edge and using an intersection algorithm for sorted lists it returns the intersection of edge lifespan and I. In SETI, TIME_JOIN retrieves the edge lifespan labels L_s and L_e and for each $[s', e'] \in I$ s.t. $\exists\ i$ s.t $\max(L_s[i],\ s') \geq \min(L_e[i],\ e')$ it returns the overlapping time instances $\{[s', e'] \cap [L_s[i], L_e[i]]\}$.

To speed-up traversal, we perform a number of pruning tests. The traversal stops when we traverse an edge that is not alive in the query interval (lines 7–10). 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 $I_e \subseteq I$ and $\mathcal{IN}(w) \sqsupseteq I$, the traversal is pruned (lines 19–23).

Indexing. The time index can be used similarly in all approaches to prune some computations. For example, for the least-k reachability query that asks whether nodes u and v are reachable in at least k time instances, we can first check using the index whether both nodes were active in at least k common time instances. If they were not active, we do not need to traverse the graph. Otherwise, we traverse the graph using a subinterval of I that contains only the instances when both nodes were active.

5 Experimental Evaluation

In this section, we present an experimental comparison of the different approaches for supporting historical traversal queries in a native graph database. We used the Neo4j[1] graph database that supports fast processing of graph data and implemented all algorithms using the Neo4j Java API.

We use two real and one synthetic dataset. In particular, we use DBLP[2] in time interval [1959, 2016] where each graph snapshot corresponds to one year. At each graph snapshot, a node represents an author and an edge a co-authorship relation between two authors in the corresponding year. We also use a FB [21] dataset which consists of 871 daily snapshots where at each snapshot a node represents a user and an edge represents a relation between two users. The synthetic dataset was generated using a preferential attachment graph generator [16], where a new snapshot is created after 10,000 nodes. The dataset characteristics are summarized in Table 1(a). The FB dataset and the default synthetic dataset are insert-only, i.e., contain no node/edge deletions.

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

Table 1. Dataset and Graph database characteristics

Dataset	# Nodes	# Edges	# Snapshots
DBLP	1,167,854	5,364,298	58
FB	61,967	905,565	871
Synthetic	1,000,000	1,999,325	100

(a) Dataset characteristics

Dataset	GDB	Size (MB)	Index Size (MB)	Time (sec)
DBLP	ME	353		39
	SETP	528.84	131.37	22
	SETI	546.55		23
FB	ME	6,000		631
	SETP	400	830	65
	SETI	31.98		33
Synthetic	ME	4,500		1,620
	SETP	513	1,700	145
	SETI	253		86

(b) Graph database size and creation time

5.1 Size and Load Time

We stored all datasets in three different database instances (GDBs) using the three different representations, namely, ME, SETP, and SETI introduced in Sect. 3.

[1] https://neo4j.com/.
[2] http://dblp.uni-trier.de.

Also, in each GDB we stored a time index on the lifespan of the nodes. Table 1(b) shows the size and construction time of each graph database instance. Multi-edge approaches use a different edge type for each time instance, which leads to larger sizes. This difference in size is more evident in the FB dataset, since most edges in the DBLP dataset have short lifespans, because many co-authorships appear only once or span very few years. To load the datasets into the graph databases we used the CSV importing system of Neo4j. Again, ME requires more time to be loaded since it has to create more edges than the other models.

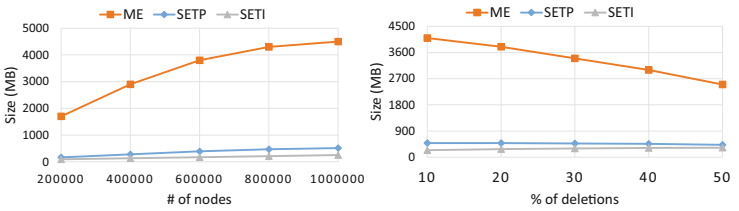


Fig. 5. Size (a) for varying number of nodes and (b) percentage of deletions

In Fig. 5(a), we report graph database sizes for varying number of nodes (and thus snapshot) using the synthetic dataset. As shown, the single-edge approaches are much smaller than the multi-edge in all cases, as expected. We also vary the percentage of edge deletes. For each edge, we randomly remove 10% to 50% of the time instances in its lifespan. Figure 5(b) presents the results. We observe that the size of ME decreases; since removing a time instance leads to less edges types. The number of removals in the lifespan (stored as lists) in SETP leads to slower size reduction. SETI size is increasing since removing time instances leads to more subintervals and thus to larger L_s and L_e lifespan structures. Overall, that single-edges are the best choice in terms of size efficiency for storing large graphs. Among them, SETI is more space-efficient, especially, when there are few subintervals in the lifespan.

5.2 Query Processing

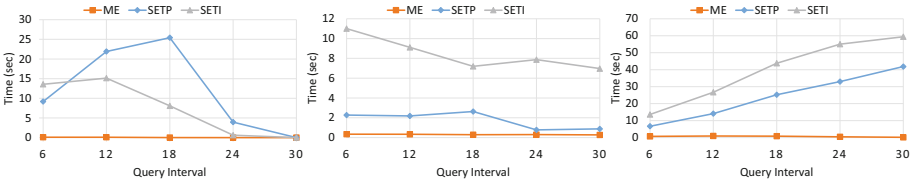
We now focus on query processing. We report the average execution time of 200 historical traversal queries where the source and target nodes are chosen uniformly at random with the restriction that both nodes are present in the graph at the beginning and the end of the query interval. For the FB and the synthetic dataset, the query interval is chosen randomly. However, in DBLP dataset which is more active in the last two decades, we use $I = [2011, 2016]$ as default query interval. For larger intervals we increase it using earlier years for starting time instances. For the *at least-k* queries we set k to be equal to $|I|/2$.

Reachability Queries. In Figs. 6 and 7, we depict the average query times for DBLP and FB. A general remark that holds independently of the graph representation model and the dataset is that disjunctive queries are faster than conjunctive queries, since they stop once an instance where the nodes are reachable is found. Conjunctive queries are in turn faster than at least-k queries, since they stop once an instance where the nodes are not reachable is found.

The main difference between the two datasets is that in DBLP edges represent co-authorships, consequently, in general, their lifespans include very few years, in most cases, just 1 or 2. In FB, lifespans are larger, and since we have no deletions, include just one interval. The ME approach is very fast for short-lived edges and is a clear winner for reachability queries in DBLP. For FB which contains a large number of multiple edge types, the response time of ME increases linearly with the size of the query interval. An exception is disjunctive reachability queries, where traversal stops once an instance where a path exists is found and ME remains competitive.

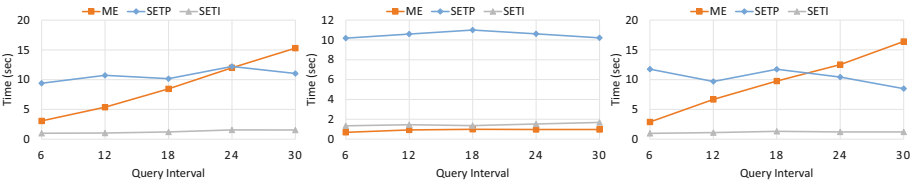
Among the single edge approaches, SETP outperforms SETI only when the lifespan includes very few time instances (as in DBLP). In this case, the time join between the lifespan and any interval is fast. Furthermore, in this case, SETI includes many small intervals. When lifespans become larger and more continuous (as in FB), SETI outperforms SETP.

To study further the effect of lifespans on query performance, we experimented using the synthetic dataset with different percentage of deletions and with a query interval of length 10 in Fig. 8. We observe that ME and SETI are competitive in conjunctive and disjunctive queries whereas in at least-k queries SETI is the winner. ME takes advantage of the use of the native TRAVERSALBFS method.



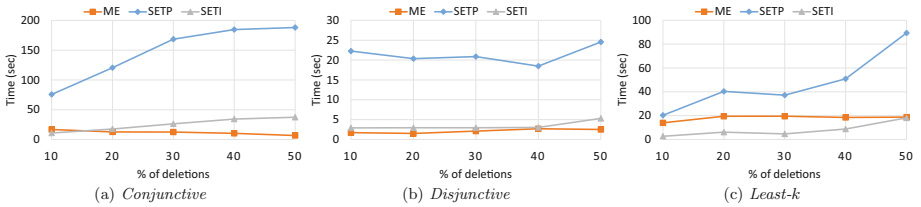
(a) *Conjunctive* (b) *Disjunctive* (c) *Least-k*

Fig. 6. Query time for historical reachability queries in DBLP



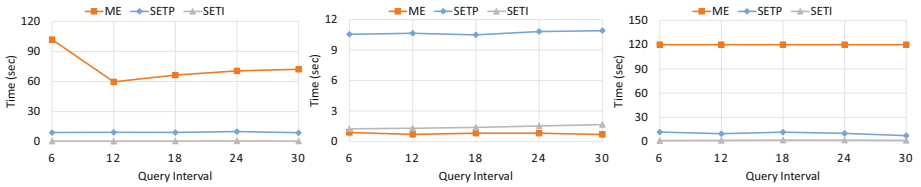
(a) *Conjunctive* (b) *Disjunctive* (c) *Least-k*

Fig. 7. Query time for historical reachability queries in FB



(a) *Conjunctive* (b) *Disjunctive* (c) *Least-k*

Fig. 8. Query time for historical reachability queries in the synthetic dataset



(a) SSP (b) ESP (c) KSP

Fig. 9. Query time for historical shortest path queries in FB

SETI performs well in all type of queries and it is starting to slow down when the percentage of deletions is getting higher and the number of intervals in the lifespan gets large.

Path Queries. We also evaluated the performance of historical path queries. ESP queries perform similar to disjunctive reachability queries, since we seek for the shortest path in the first instance when the two nodes are connected. However, in case of SSP and KSP we need to locate the shortest among paths that exist in all or in at least-k instances. We experimented with a large number of random pair of nodes and observed that in DBLP no paths that connect these pairs exist in more than 6 time instances. Furthermore, in most cases, these paths existed in just a single instance. In Fig. 9, we report the average time for shortest path queries in FB. The processing in ME is costly since for each traversed edge that connects u to v the traversal algorithm has to check if there are also other type of edges that refer to all (or k) time instances that u to v. Thus, we set a limit of 120 s for each path query type. KSP queries in ME exceed the time limit for computing a solution. In general, SETI is the fastest one and SETP comes second in SSP and KSP queries, since they traverse a small number of queries compared to multi-edge and the edge lifespan verification in the given interval is performed fast.

Time Index. Finally, we ran the same historical traversal queries in DBLP and FB datasets without using the time index and we observed that in general the time index improves query performance. Due to space constraints, we only depict the change in performance for conjunctive queries in Fig. 10(a)(b). In particular, in DBLP dataset we observe high performance as long as the query interval is

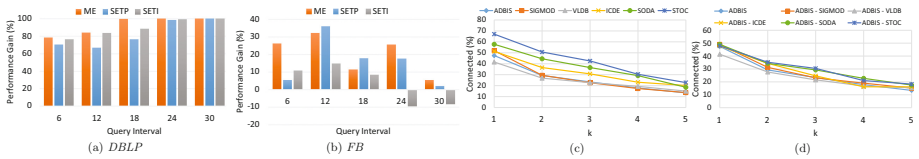


Fig. 10. (a)(b) Time index performance boost for conjunctive queries and (c)(d) percentage of connected pair of nodes in various conferences

increasing since there are not many connected pairs in all time instances and thus indexing returns the negative answers very fast. However, in FB dataset where there are nodes that are connected in whole interval even for larger ones, we notice that indexing is more helpful in ME and SETP since we do not pay the cost for traversing the graph for pairs that are not connected. SETI performance in FB does not increase very much since traversal algorithms run very fast by pruning edges that are not active in the interval. The same trend is observed in historical path queries and thus results are omitted.

5.3 Case Study

In this study, we use historical queries to study connectivity between authors at difference conferences in DBLP. We selected 4 database (ADBIS, SIGMOD, VLDB, ICDE) and 2 theory (SODA, STOC) conferences. For each conference, we randomly selected 500 pair of nodes representing authors that have at least one publication in the conference and examined whether they are reachable in at least k years in the interval [1959, 2016]. We depict the results in Fig. 10(c) where we observe that theory conferences have the most reachable pairs of nodes which indicates that they consist of more well-connected communities compared to database conferences. As expected, the percentage of nodes that are reachable decreases as k increases. We also conducted a second study to show connectivity between ADBIS authors and authors in the other 5 conferences. As show in Fig. 10(d), somehow surprisingly ADBIS authors are more connected with authors in the theory conferences than with authors in the database conferences. Not surprisingly, connectivity between authors of the same conference is larger than connectivity among ADBIS and other conferences.

6 Conclusions

In this paper, we study the problem of storing and querying the history of an evolving graph in a native graph database. We have proposed different approaches for storing such graphs based on associating with each node and edge a lifespan, i.e., a set of time intervals indicating when they were valid. We have also proposed algorithms for processing various types of traversal queries using the proposed storage models. For very short-lived edges, using multiple edges to represent lifespans, one for each time instance, seems to work well by

taking advantage of the built-in traversal methods of the native graph databases. However, for all other cases, using an interval-based approach to represent lifespans proves more efficient both processing and storage wise. There are many possible directions for future work. One is to extend historical queries to include time-varying node and edges labels, that is labels, that change over time. Another direction is to provide support for historical graph queries inside the native graph database.

References

1. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: WWW, pp. 237–248 (2014)
2. Backstrom, L., Huttenlocher, D.P., Kleinberg, J.M., Lan, X.: Group formation in large social networks: membership, growth, and evolution. In: KDD, pp. 44–54 (2006)
3. Bahmani, B., Chowdhury, A., Goel, A.: Fast incremental and personalized pagerank. PVLDB **4**(3), 173–184 (2010)
4. Cattuto, C., Quaggiotto, M., Panisson, A., Averbuch, A.: Time-varying social networks in a graph database: a Neo4j use case. In: GRADES, p. 11 (2013)
5. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: EuroSys, pp. 85–98 (2012)
6. Durand, G.C., Pinnecke, M., Broneske, D., Saake, G.: Backlogs and interval timestamps: building blocks for supporting temporal queries in graph databases. In: EDBT/ICDT Workshops (2017)
7. Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., Chen, E.: Chronos: a graph engine for temporal graph analysis. In: EuroSys, p. 1 (2014)
8. Huang, H., Song, J., Lin, X., Ma, S., Huai, J.: Tgraph: a temporal graph data management system. In: CIKM, pp. 2469–2472 (2016)
9. Huo, W., Tsotras, V.J.: Efficient temporal shortest path queries on evolving social graphs. In: SSDBM, p. 38 (2014)
10. Jensen, C.S., Snodgrass, R.T.: Temporal element. In: Liu, L., Tamer Özsu, M. (eds.) Encyclopedia of Database Systems, p. 2966. Springer, Heidelberg (2009). doi:10.1007/978-0-387-39940-9_1419
11. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: ICDE, pp. 997–1008 (2013)
12. Koloniari, G., Souravlias, D., Pitoura, E.: On graph deltas for historical queries. In: WOSS (2012)
13. Labouseur, A.G., Birnbaum, J., Olsen Jr., P.W., Spillane, S.R., Vijayan, J., Hwang, J.H., Han, W.S.: The G* graph database: efficiently managing large distributed dynamic graphs. Distrib. Parallel Databases **33**, 479–514 (2014)
14. Labouseur, A.G., Olsen, P.W., Hwang, J.H.: Scalable and robust management of dynamic graph data. In: VLDB, pp. 43–48 (2013)
15. Leskovec, J., Kleinberg, J.M., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. In: KDD, pp. 177–187 (2005)
16. Newman, M.E.J.: The structure and function of complex networks. SIAM Rev. **45**(2), 167–256 (2003)

17. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. PVLDB **4**(11), 726–737 (2011)
18. Semertzidis, K., Lillis, K., Pitoura, E.: Timereach: historical reachability queries on evolving graphs. In: EDBT, pp. 121–132 (2015)
19. Semertzidis, K., Pitoura, E.: Durable graph pattern queries on historical graphs. In: ICDE, pp. 541–552 (2016)
20. Semertzidis, K., Pitoura, E.: Time traveling in graphs using a graph database. In: EDBT/ICDT Workshops (2016)
21. Viswanath, B., Mislove, A., Cha, M., Gummadi, P.K.: On the evolution of user interaction in Facebook. In: WOSN, pp. 37–42 (2009)