Efficient Snapshot Retrieval over Historical Graph Data

Udayan Khurana, Amol Deshpande

University of Maryland, College Park
{udayan,amol}@cs.umd.edu

***Abstract*—We present a distributed graph database system to manage historical data for large evolving information networks, with the goal to enable temporal and evolutionary queries and analysis. The cornerstone of our system is a novel, user-extensible, highly tunable, and distributed hierarchical index structure called *DeltaGraph*, that enables compact recording of the historical network information, and that supports efficient retrieval of historical graph snapshots for single-site or parallel processing. Our system exposes a general programmatic API to process and analyze the retrieved snapshots. Along with the original graph data, DeltaGraph can also maintain and index *auxiliary* information; this functionality can be used to extend the structure to efficiently execute queries like *subgraph pattern matching* over historical data. We develop analytical models for both the storage space needed and the snapshot retrieval times to aid in choosing the right construction parameters for a specific scenario. We also present an in-memory graph data structure called *GraphPool* that can maintain hundreds of historical graph instances in main memory in a non-redundant manner. We present a comprehensive experimental evaluation that illustrates the effectiveness of our proposed techniques at managing historical graph information.**

I. Introduction

In recent years, we have witnessed an increasing abundance of observational data describing various types of information networks, including social networks, biological networks, citation networks, financial transaction networks, communication networks, to name a few. There is much work on analyzing such networks to understand various social and natural phenomena like: *"how the entities in a network interact"*, *"how information spreads"*, *"what are the most important (*central*) entities"*, and *"what are the key building blocks of a network"*. With the increasing availability of the digital trace of such networks *over time*, the topic of network analysis has naturally extended its scope to *temporal* analysis of networks, which has the potential to lend much better insights into various phenomena, especially those relating to the temporal or evolutionary aspects of the network. For example, we may want to know: *"which analytical model best captures a network's evolution"*, *"how information spreads over time"*, *"who are the people with the steepest increase in centrality measures over a period of time"*, *"what is the average monthly density of a network since 1997"*, *"how the clusters in a network evolve over time"* etc. Historical queries like, *"who had the highest PageRank centrality in a citation network in 1960"*, *"which year amongst 2001 and 2004 had the smallest network diameter"*, *"how many new triangles have been formed in the network over the last year"*, also involve the temporal aspect of the network. More generally a network analyst may want to process a network's historical trace in different, usually unpredictable ways to gain insights into various phenomena. There is also interest in visualizations over temporal graphs [1].

To support a broad range of network analysis tasks, we require a graph[1] data management system at the backend capable of low-cost storage and efficient retrieval of the historical network information, in addition to maintaining the *current* state of the network for updates and other queries, temporal or otherwise. However, the existing solutions for graph data management lack adequate techniques for temporal annotation, or for storage and retrieval of large scale historical changes on the graph. In this paper, we present the design of a graph data management system that we are building to provide support for executing temporal analysis queries and historical queries over large-scale evolving information networks.

Our primary focus in this paper is on efficiently supporting **snapshot retrieval queries** where the goal is *to retrieve in memory one or more historical snapshots of the information network as of specified time points*. The typically unpredictable, iterative, and procedural nature of network analysis makes this perhaps the most important type of query that needs to be supported. We assume there is enough memory to hold the retrieved snapshots in memory (we discuss below how we exploit overlap in the retrieved snapshots to minimize the memory requirements); we allow the snapshots to be retrieved in a **partitioned** fashion across a set of machines in parallel to handle very large networks. This design decision was motivated by both the current hardware trends and the fact that, most network analysis tasks tend to access the underlying network in unpredictable ways, leading to unacceptably high penalties if the data does not fit in memory. Most current large-scale graph analysis systems, including Pregel [18], Giraph[2], Trinity [27], Cassovary (Twitter graph library)[3], Pegasus [12], load the entire graph into memory prior to execution.

The cornerstone of our system is a novel hierarchical index structure called **DeltaGraph**. A DeltaGraph is a rooted, directed graph whose lowest level corresponds to the snapshots

[1]We use the terms *graph* and *network* interchangeably in this paper.
[2]http://giraph.apache.org
[3]https://github.com/twitter/cassovary

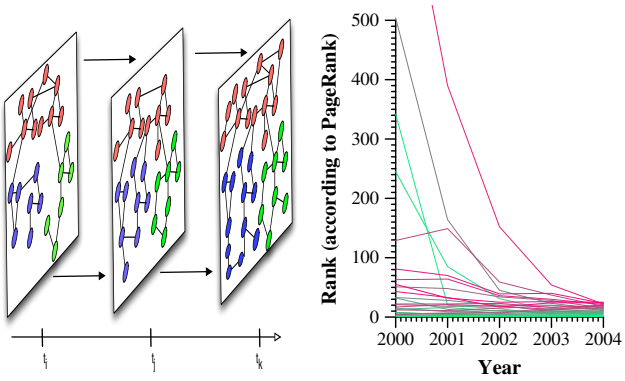


Fig. 1. Dynamic network analysis (e.g., understanding how "communities" evolve in a social network, how centrality scores change in co-authorship networks, etc.) can lend important insights into social, cultural, and natural phenomena. The right plot was constructed using our system over the DBLP network, and shows the evolution of the nodes ranked in top 25 in 2004.

of the network over time (that are not explicitly stored), and the interior nodes correspond to graphs constructed by combining the lower level graphs (these are typically not valid snapshots as of any specific time point). The information stored with each edge, called **edge deltas**, is sufficient to construct the graph corresponding to the target node from the graph corresponding to the source node, and thus a specific snapshot can be created by traversing any path from the root to the snapshot. While conceptually simple, DeltaGraph is a powerful, general, and tunable index structure that enables trading off different resources and user requirements as per a specific application's need, both at construction time and at run-time. Portions of the DeltaGraph can be pre-fetched and **materialized**, allowing us to trade increased memory utilization for reduced query times. DeltaGraph is also **extensible**, providing a user the opportunity to define additional indexes to be created and maintained in order to efficiently execute specific queries (e.g., subgraph pattern matching, reachability, etc.) over the historical graph data. Finally, DeltaGraph utilizes several other optimizations including a **column-oriented storage** to minimize the data that needs to be fetched to answer a query, and **multi-query optimization** to simultaneously retrieve many snapshots.

DeltaGraph naturally enables distributed storage and processing to scale to very large graphs. The edge deltas can be stored in a distributed fashion through use of horizontal partitioning, and the historical snapshots can be loaded parallely onto a set of machines in a partitioned fashion; in general, the two partitionings need not be aligned, but for computational efficiency, we currently require that they be aligned. Horizontal partitioning also results in lower snapshot retrieval latencies since the different deltas needed for reconstruction can be fetched in parallel.

The second key component of our system is an in-memory data structure called **GraphPool**. A typical network evolution query may require analyzing 100's of snapshots from the history of a graph. Maintaining these snapshots in memory independently of each other would likely be infeasible. The GraphPool data structure exploits the commonalities in the snapshots that are currently in memory, by overlaying them on a single graph data structure (typically a *union* of all the snapshots in memory). GraphPool also employs several optimizations to minimize the amount of work needed to incorporate a new snapshot and to clean up when a snapshot is purged after the analysis has completed.

We have built a prototype implementation of our system in Java, using the Kyoto Cabinet[4] disk-based key-value store as the back-end engine to store the DeltaGraph components (in the distributed case, we run one instance on each machine). Our design decision to use a key-value store at the back-end was motivated by the flexibility, the fast retrieval times, and the scalability afforded by such systems; since we only require a simple *get/put* interface from the storage engine, we can easily plug in other cloud-based, distributed key-value stores like HBase[5]. Our comprehensive experimental evaluation shows that our system can retrieve historical snapshots containing up to millions of nodes and edges in several 100's of milliseconds or less, often an order of magnitude faster than prior techniques like *interval trees*, and that the execution time penalties of our in-memory data structure are minimal.

Finally, we note that our proposed techniques are general and can be easily extended for efficient snapshot retrieval in temporal relational databases as well.

Outline: We begin with a discussion of the prior work (Section II). We then discuss the key components of the system, the data model, and present the high level system architecture (Section III). Then, we describe the DeltaGraph structure in detail (Section IV), and develop analytical models for the storage space and snapshot retrieval times (Section V). We then briefly discuss GraphPool (Section VI). Finally, we present the results of our experimental evaluation (Section VII).

II. Related Work

We only discuss the most related work here and refer the reader to the extended version of the paper for a more detailed discussion [13]. There has been an increasing interest in dynamic network analysis over the last decade, fueled by the increasing availability of large volumes of temporally annotated network data. Many works have focused on designing analytical models that capture how a network evolves (see, e.g., [16], [14]). There is also much work on understanding how communities or graph structures evolve, identifying key individuals, locating hidden groups in dynamic networks, and visualizing temporal evolution [5], [30], [10], [20], [4], [1], [24], [21]. Our goal in this work is to build a graph data management system that can efficiently and scalably support these types of dynamic network analysis tasks over large volumes of data in real-time.

There is a vast body of literature on *temporal relational databases*, starting with the early work in the 80's on developing temporal data models and temporal query languages. We won't attempt to present an exhaustive survey of that work, but instead refer the reader to several surveys and books on this

[4]http://fallabs.com/kyotocabinet
[5]http://hbase.apache.org

topic [22], [31], [8], [29], [25]. The most basic concepts that a relational temporal database is based upon are **valid time** and **transaction time**, considered orthogonal to each other. Under that nomenclature, our data management system is based on valid time. From a querying perspective, both valid-time and transaction-time databases can be treated as simply collections of intervals. Salzberg and Tsotras [25] present a comprehensive survey of indexing structures for temporal databases. They also present a classification of different queries that one may ask over a temporal database. Under their notation, our focus in this work is on the **valid timeslice** query, where the goal is to retrieve all the entities and their attribute values that are valid as of a specific time point. We discuss the related work on snapshot retrieval queries in more detail in Section IV-A.

There has been resurgence of interest in general-purpose graph data management systems in both academia and industry. Several commercial and open-source graph management systems are being actively developed (e.g., Neo4j[6], GBase[7], Pregel [18], Giraph, Trinity [27], Cassovary, Pegasus [12]). There is much ongoing work on efficient techniques for answering various types of queries over graphs and on building indexing structures for them. However, we are not aware of any graph data management system that focuses on optimizing snapshot retrieval queries over *historical* traces, and on supporting rich temporal analysis of large networks.

There is also prior work on temporal RDF data and temporal XML Data. Several works (e.g., [23], [32]) have considered the problems of subgraph pattern matching or SPARQL query evaluation over temporally annotated RDF data. There is also much work on version management in XML data stores and scientific datasets [7], [15], [19]. Ghandeharizadeh et al. [9] provide a formalism on deltas, which includes a *delta arithmetic*. All these approaches assume unique node identifiers to merge deltas with deltas or snapshots. Buneman et al. [7] propose merging all the versions of the database into one single hierarchical data structure for efficient retrieval. In a recent work, Seering et al. [26] present a disk-based versioning system that uses efficient delta encoding to minimize space consumption and retrieval time in array-based systems. Lomet et al. [17] show how to integrate a temporal index (TSB-tree) into SQL Server. However, none of that prior work focuses on snapshot retrieval in general graph databases, or proposes techniques that can flexibly exploit the memory-resident information.

III. Overview

We begin with briefly describing our graph data model, the system architecture, and different types of snapshot retrieval queries that we support.

A. Graph Data Model

The most basic model of a graph over a period of time is as a collection of *graph snapshots*, one corresponding to each time instance (we assume discrete time). Each such graph snapshot contains a set of nodes and a set of edges. The nodes and edges are assigned unique ids at the time of their creation, which are not re-assigned after deletion of the components. A node or an edge may be associated with a list of attribute-value pairs; the list of attribute names is not fixed a priori and new attributes may be added at any time. Additionally an edge contains the information about whether it is a directed edge or an undirected edge.

We define an *event* as the record of an atomic activity in the network. An event could pertain to either the creation or deletion of an edge or node, or change in an attribute value of a node or an edge. Alternatively, an event can express the occurrence of a *transient* edge or node that is valid only for that time instance instead of an interval (e.g., a "message" from a node to another node). Being atomic refers to the fact that the activity can not be logically broken down further into smaller events. Hence, an event always corresponds to a single timepoint. So, the valid time interval of an edge, $[t_s, t_e]$, is expressed by two different events, edge addition and deletion events at t_s and t_e respectively. The exact contents of an event depend on the event type; below we show examples of a new edge event (NE), and an update node attribute event (UNA).

```
(a) {NE, N:23, N:4590, directed:no, 11/29/03 10:10}
(b) {UNA, N:23, 'job', old:'..', new:'..', 11/29/07 17:00}
```

We treat events as bidirectional, i.e., they could be applied to a database snapshot in either direction of time. For example, say that at times t_{k-1} and t_k, the graph snapshots are S_{k-1} and S_k respectively. If E is the set of all events at time t_k, we have that:

$$S_k = S_{k-1} + E, \qquad S_{k-1} = S_k - E$$

where the $+$ and $-$ operators denote application of the events in E in the forward and the backward direction. A list of chronologically organized events is called an *eventlist*.

B. System Overview

Figure 2 shows a high level overview of our system and its key components. At a high level, there are multiple ways

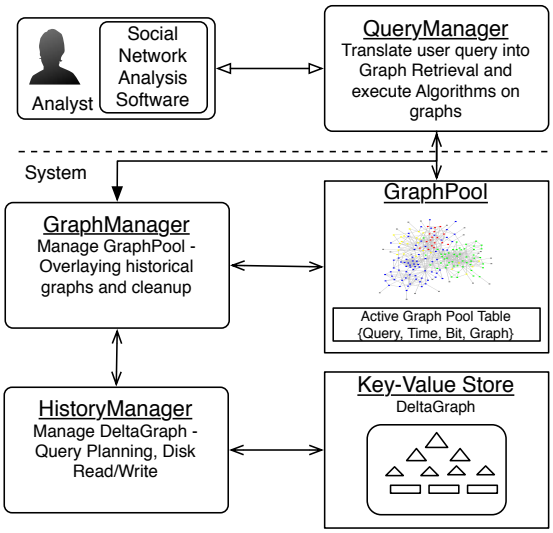


Fig. 2. Architecture of our system: our focus in this work is on the components below the dashed line.

[6]http://www.neo4j.org
[7]http://www.graphbase.net

that a user or an application may interact with a historical graph database. Given the wide variety of network analysis or visualization tasks that are commonly executed against an information network, we expect a large fraction of these interactions will be through a **programmatic API** where the user or the application programmer writes her own code to operate on the graph (as shown in the figure). Such interactions result in what we call *snapshot* queries being executed against the database system. Executing such queries is the primary focus of this paper, and we further discuss these types of queries below. In ongoing work, we are also working on developing a high-level declarative query language (similar to TSQL [29]) and query processing techniques to execute such queries against our database. As a concrete example, an analyst who may have designed a new network evolution model and wants to see how it fits the observed data, may want to retrieve a set of historical snapshots and process them using the programmatic API. On the other hand, a declarative query language may better fit the needs of a user interested in searching for a temporal pattern (e.g., *find nodes that had the fastest growth in the number of neighbors since joining the network*). Next, we briefly discuss snapshot queries and the key components of the system.

1) Snapshot Queries: We differentiate between a **single-point snapshot query** and a **multipoint snapshot query**. An example of the first query is: "Retrieve the graph as of January 2, 1995". On the other hand, a multipoint snapshot query requires us to simultaneously retrieve multiple historical snapshots (e.g, "Retrieve the graphs as of every Sunday between 1994 to 2004"). We also support more complex snapshot queries where a *TimeExpression* or a *time interval* is specified instead. Any snapshot query can specify whether it requires only the structure of the graph, or a specified subset of the node or edge attributes, or all attributes. Specifically, the following is a list of some of the retrieval functions that we support in our programmatic API.

GetHistGraph(Time *t*, String *attr_options*): retrieve the graph as of time t, with *attr_options* specifying the attributes to fetch. For example, to fetch all node attributes except *salary*, and only the edge attribute *name*, we would use:

attr_options = "+node:all-node:salary+edge:name"

GetHistGraphs(List<Time> *t_list*, String *attr_options*): fetch multiple graphs, with *t_list* specifying a list of time points.

GetHistGraph(TimeExpression *tex*, String *attr_options*): fetch a hypothetical graph using a multinomial Boolean expression over time points. For example, the expression $(t_1 \wedge \neg t_2)$ specifies the components of the graph that were valid at time t_1 but not at time t_2.

GetHistGraphInterval(Time t_s, Time t_e, String *attr_options*): retrieve a graph consisting of all the elements that were added during the time interval $[t_s, t_e)$.

Eventually, our goal is to support **Blueprints**[8], a collection of interfaces analogous to JDBC but for graph data (we currently support a subset). Blueprints is a generic graph Java API that already binds to various graph database backends (e.g., Neo4j), and many graph processing and programming frameworks are built on top of it (e.g., Gremlin, a graph traversal language[9]; Furnace, a graph algorithms package[10]; etc.). By supporting the Blueprints API, we immediately enable use of many of these already existing toolkits. The (Java) code snippet below shows an example program that retrieves several graphs, and operates upon them.

```
GraphManager gm = new GraphManager(...);
gm.loadDeltaGraphIndex(...); // Load index information
// Retrieve graph structure as of Jan 2, 1985, with node names
HGraph h1 = gm.GetHistGraph("1/2/1985", "+node:name");
// Traverse the graph...
List<HNode> nodes = h1.getNodes();
List<HNode> neighborList = nodes.get(0).getNeighbors();
HEdge ed = h1.getEdge(nodes.get(0), neighborList.get(0)); ...
// Retrieve graphs (structures only) on Jan 2 of 1986 and 1987
listOfDates.add("1/2/1986");
listOfDates.add("1/2/1987");
List<HGraph> h2 = gm.getHistGraphs(listOfDates, "");
```

2) Key Components: There are two key data structure components of our system.

1. **GraphPool** (Section VI) is an in-memory data structure that can store multiple graphs together in a compact way by overlaying the graphs on top of each other. At any time, the GraphPool contains: (1) the *current graph* that reflects the current state of the network, (2) the *historical snapshots*, retrieved from the past using the commands above and possibly modified by an application program, and (3) *materialized graphs*, which are graphs that correspond to interior or leaf nodes in the DeltaGraph, but may not correspond to any valid graph snapshot (Section IV-E).
2. **DeltaGraph** (Section IV) is an index structure that stores the historical network data using a hierarchical index structure over *deltas* and *leaf-level eventlists* (called *leaf-eventlists*). To execute a snapshot retrieval query, a set of appropriate deltas and leaf-eventlists is fetched and the resulting graph snapshot is overlaid on the existing set of graphs in the GraphPool. The structure of the DeltaGraph itself, called **DeltaGraph skeleton**, is maintained as a weighted graph in memory (it contains statistics about the deltas and eventlists, but not the actual data). The skeleton is used during query planning to choose the optimal set of deltas and eventlists for a given query.

The data structures are managed and maintained by several system components. *HistoryManager* deals with the construction of the DeltaGraph, plans how to execute a singlepoint or a multipoint snapshot query, and reads the required deltas and eventlists from the disk. *GraphManager* is responsible for managing the GraphPool data structure, including the overlaying of deltas and eventlists, bit assignment, and post-

[8]http://github.com/tinkerpop/blueprints

[9]http://github.com/tinkerpop/gremlin/wiki

[10]http://github.com/tinkerpop/furnace/wiki

query clean up. Finally, the *QueryManager* manages the interface with the user or the application program, and extracts a snapshot query to be executed against the DeltaGraph. One of its functions is to translate any explicit references (e.g., *user-id*) from the query to the corresponding internal-id and vice-versa for the final result, using a lookup table. As discussed earlier, such a component is highly application-specific, and we do not discuss it further in this paper.

3) Distributed Deployment: In a distributed deployment, DeltaGraph and GraphPool are both partitioned across a set of machines by partitioning the node ID space, and assigning each partition to a separate machine (Section IV-F). The partitioning used for storage can be different from that used for retrieval and processing; however, for minimizing wasted network communication, it would be ideal for the two partitionings to be aligned so that multiple DeltaGraph partitions may correspond to a single GraphPool partition, but not vice versa. Snapshot retrieval on each machine is independent of the others, and requires no network communication among those. Once the snapshots are loaded into the GraphPool, any distributed programming framework can be used on top; we have implemented an **iterative vertex-based message-passing system** analogous to Pregel [18].

For clarity, we assume a single-site deployment (i.e., no horizontal partitioning) in most of the description that follows.

IV. DeltaGraph: Indexing Historical Graph Data

We begin with discussing previously proposed techniques for supporting snapshot queries, and why they do not meet our needs. We then present the DeltaGraph data structure.

A. Prior Techniques and Limitations

An optimal (within constant factors) solution to answering snapshot retrieval queries is the **external interval tree** [3]. It uses optimal space on disk and supports updates in optimal (logarithmic) time. **Segment trees** [6] can also be used to solve this problem, but may store some intervals in a duplicated manner and hence use more space. Tsotras and Kangelaris [33] present **snapshot index**, an I/O optimal solution to the problem for transaction-time databases. Salzberg and Tsotras [25] also discuss two extreme approaches to supporting snapshot retrieval queries, called **Copy** and **Log** approaches. In the Copy approach, a snapshot of the database is stored at each transaction state, the primary benefit being fast retrieval times; however the space requirements make this approach infeasible in practice. The other extreme approach is the Log approach, where only and all the changes are recorded to the database, annotated by time. While this approach is space-optimal and supports $O(1)$-time updates (for transaction-time databases), answering a query may require scanning the entire list of changes and takes prohibitive amount of time. A mix of those two approaches, called **Copy+Log**, where a subset of the snapshots are explicitly stored, is often a better idea.

We found these (and other prior) approaches to be insufficient and inflexible for our needs for many reasons. (1) They do not efficiently support multipoint queries that we expect to be very commonly used in evolutionary analysis, that need to be optimized by avoiding duplicate reads and repeated processing of the events. (2) To cater to the needs of a variety of different applications, we need the index structure to be highly tunable, to allow trading off different resources and user requirements (including memory, disk usage, and query latencies). Ideally we would also like to control the distribution of average snapshot retrieval times over the history, i.e., we should be able to reduce the retrieval times for more recent snapshots at the expense of increasing it for the older snapshots (while keeping the utilization of the other resources the same), or vice-versa. (3) For achieving low latencies, the index structure should support flexible pre-fetching of portions of the index into memory and should avoid processing any events that are not needed by the query (e.g., if only the network structure is needed, then we should not have to process any events pertaining to the node or edge attributes). (4) Finally, we would like the index structure to be able to support different persistent storage options, ranging from a hard disk to the cloud; most of the previously proposed index structures are optimized primarily for disks.

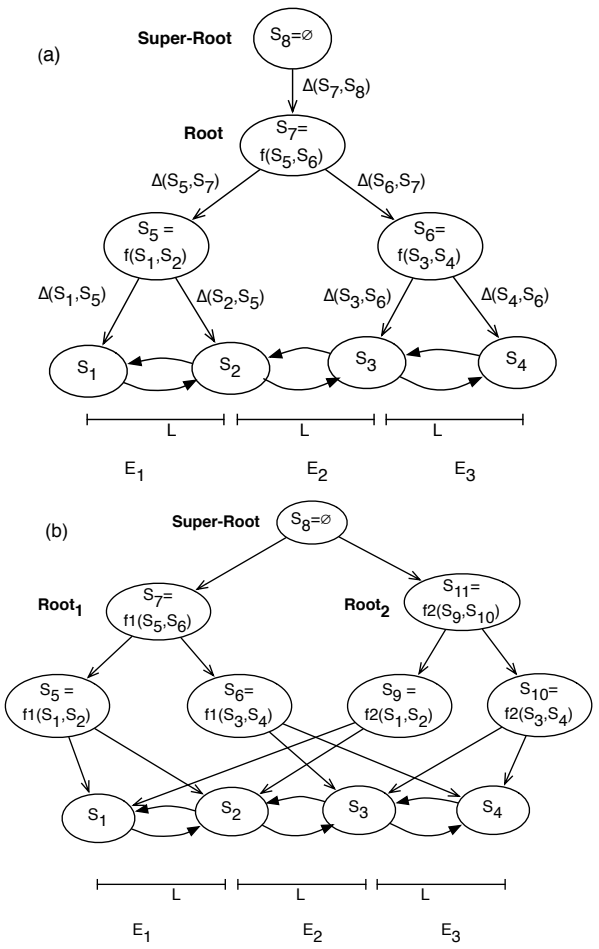
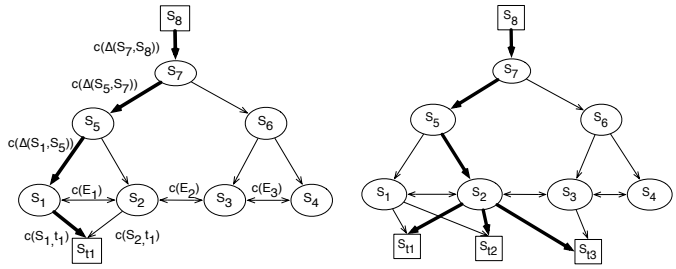


Fig. 3. DeltaGraphs with 4 leaves, leaf-eventlist size L, arity 2. $\Delta(S_i, S_j)$ denotes delta needed to construct S_i from S_j.

B. DeltaGraph Overview

Our proposed index data structure, DeltaGraph, is a directed graphical structure that is largely hierarchical, with



(a) Singlepoint query t_1 (b) Multipoint query $\{t_1, t_2, t_3\}$

Fig. 4. Example plans for singlepoint and multipoint retrieval on the DeltaGraph shown in Figure 3(a).

the lowest level of the structure corresponding to equi-spaced historical snapshots of the network (equal spacing is not a requirement, but simplifies analysis). Figure 3(a) shows a simple DeltaGraph, where the nodes $S_1, \ldots, S_4$ correspond to four historical snapshots of the graph, spaced L events apart. We call these nodes *leaves*, even though there are bidirectional edges between these nodes as shown in the figure. The interior nodes of the DeltaGraph correspond to graphs that are constructed from its children by applying what we call a **differential function**, denoted $f()$. For an interior node S_p with children $S_{c_1}, \ldots, S_{c_k}$,[11] we have that $S_p = f(S_{c_1}, \ldots, S_{c_k})$. The simplest differential function is perhaps the **Intersection** function. We discuss other functions in Section V.

The graphs S_p are not explicitly stored in the DeltaGraph. Rather we only store the *delta* information with the edges. Specifically, the directed edge from S_p to S_{c_i} is associated with a delta $\Delta(S_{c_i}, S_p)$ that allows construction of S_{c_i} from S_p. It contains the elements that should be deleted from S_p (i.e., $S_p - S_{c_i}$) and those that should be added to S_p (i.e., $S_{c_i} - S_p$). The bidirectional edges among the leaves also store similar deltas; here the deltas are simply the eventlists (denoted E_1, E_2, E_3 in Figure 3), called **leaf-eventlists**. For a leaf-eventlist E, we denote by $[E^{start}, E^{end})$ the time interval that it corresponds to. For convenience, we add a special root node, called **super-root**, at the top of the DeltaGraph that is associated with a null graph (S_8 in Figure 3). For convenience, we call the children of the super-root as **roots**.

A DeltaGraph can simultaneously have multiple hierarchies that use different differential functions (Figure 3(b)); this can be used to improve query latencies at the expense of higher space requirement.

The deltas and the leaf-eventlists are given unique ids in the DeltaGraph structure, and are stored in a **columnar fashion**, by separating out the structure information from the attribute information. For simplicity, we assume here a separation of a delta Δ (similarly an eventlist E) into three components: (1) Δ_{struct} (E_{struct}), (2) $\Delta_{nodeattr}$ ($E_{nodeattr}$), and (3) $\Delta_{edgeattr}$ ($E_{edgeattr}$). For a leaf-eventlist E, we have an additional component, $E_{transient}$, where the transient events are stored.

Finally, the deltas and the leaf-eventlists are partitioned and stored in a distributed key-value store. The *node_id* space is partitioned using a hash function for this purpose, and the information associated with an edge is stored in partitions corresponding to both its endpoints (information is not duplicated if the endpoints are in the same partition). The key used for the key-value store is $\langle$*partition_id, delta_or_elist_id,* $c\rangle$, where $c \in \{\Delta_{struct}, \Delta_{nodeattr}, \ldots, E_{transient}\}$ specifies which of the components is being fetched or stored, *partition_id* specifies the partition, and *delta_or_elist_id* specifies the unique id corresponding to the delta or the eventlist.

C. Singlepoint Snapshot Queries

Given a singlepoint snapshot query at time t_1, there are many ways to answer it from the DeltaGraph. Let E denote the leaf-eventlist such that $t_1 \in [E^{start}, E^{end})$ (found through a *binary search* at the leaf level). Any (directed) path from the super-root to the two leaves adjacent to E represents a valid solution to the query. Hence we can find the optimal solution by finding the path with the lowest weight, where the weight of an edge captures the cost of reading the associated delta (or the required subset of it), and applying it to the graph constructed so far. We approximate this cost by using the size of the delta retrieved as the weight. Note that, each edge is associated with three or more weights, corresponding to different *attr_options*. In the distributed case, we have a set of weights for each partition. We also add a new virtual node (node S_{t_1} in Figure 4(a)), and add edges to it from the adjacent leaves as shown in the figure. The weights associated with these two edges are set by estimating the portion of the leaf-eventlist E that must be processed to construct S_{t_1} from those leaves.

We use the standard Dijkstra's shortest path algorithm to find the optimal solution for a specific singlepoint query, using the appropriate weights. The shortest path thus obtained gives the minimal deltas and events to be fetched, considering the different $attr\_options$ for each query, **memory materialization** (discussed below) and other frequent changes to the DeltaGraph over time.

D. Multipoint Snapshot Queries

Similarly to singlepoint snapshot queries, a multipoint snapshot query can be reduced to finding a *Steiner tree* in a weighted directed graph. We illustrate this through an example. Consider a multipoint query over three timepoints t_1, t_2, t_3 over the DeltaGraph shown in Figure 3(a). We first identify the leaf-level eventlists that contain the three time points, and add virtual nodes S_{t1}, S_{t2}, S_{t3} as shown in Figure 4(b). The optimal solution to construct all three snapshots is then given by the lowest-weight Steiner tree that connects the super-root and the three virtual nodes (using appropriate weights depending on the attributes that need to be fetched). A possible Steiner tree is depicted in the figure using thicker edges. As we can see, the optimal solution to the multipoint query may not use the optimal solutions for each of the constituent singlepoint queries. Finding the lowest weight Steiner tree is unfortunately NP-Hard (and much harder for directed graphs vs undirected graphs), and we instead use the standard 2-approximation for undirected Steiner trees for that purpose.

[11] We abuse the notation somewhat to let S_p denote both the interior node and the graph corresponding to it.

We first construct a complete undirected graph over the set of nodes comprising the root and the virtual nodes, with the weight of an edge between two nodes set to be the weight of the shortest path between them in the skeleton. We then compute the minimum spanning tree over this graph, and "unfold" it to get a Steiner tree over the original skeleton. This algorithm does not work for general directed graphs, however we can show that, because of the special structure of a DeltaGraph, it not only results in valid Steiner trees, but retains the 2-approximation guarantee as well. We omit the details for lack of space.

Aside from multipoint snapshot queries, this technique is also used for queries asking for a graph valid for a composite TimeExpression, which we currently execute by fetching the required snapshots into memory and then operating upon them to find the components that satisfy the TimeExpression.

E. Memory Materialization

For improving query latencies, some nodes in the DeltaGraph are typically pre-fetched and materialized in memory. In particular, the highest levels of the DeltaGraph should be materialized, and further, the "rightmost" leaf (that corresponds to the current graph or a recent graph) should also be considered as materialized. The task of materializing one or more DeltaGraph nodes is equivalent to running a singlepoint or a multipoint snapshot retrieval query, and we can use the algorithms discussed above for that purpose. After a node is materialized, we modify the in-memory DeltaGraph skeleton by adding a directed edge with weight 0 from the super-root to that node. Any further snapshot retrieval queries will automatically benefit from the materialization.

The option of memory materialization enables fine-grained runtime control over the query latencies and the memory consumption, without the need to reconstruct the DeltaGraph. For instance, if we know that a specific analysis task may access snapshots from a specific period, then we can materialize the lowest common ancestor of the snapshots from that period to reduce the query latencies. One extreme case is what we call **total materialization**, where all the leaves are materialized in memory. This reduces to the Copy+Log approach with the difference that the snapshots are stored in memory in an overlaid fashion (in the GraphPool). For mostly-growing networks (that see few deletions), such materialization can be done cheaply resulting in very low query latencies.

F. DeltaGraph Construction

We first present a *batch* construction algorithm that operates on a historical trace of a network, and then present our approach to keep the DeltaGraph up-to-date as new events arrive. Besides the graph itself, represented as a list of all events in a chronological order, E, the batch construction algorithm accepts four parameters: (1) L, the size of a leaf-level eventlist; (2) k, the arity of the graph; (3) $f()$, the differential function that computes a combined delta from a given set of deltas; and (4) a partitioning of the node ID space. The DeltaGraph is constructed in a bottom-up fashion, similar to how a *bulkloaded B+-Tree* is constructed. We scan E from the beginning, creating the leaf snapshots and corresponding eventlists (containing L events each). When k of the snapshots are created, a parent interior node is constructed from those snapshots. Then the edge deltas are created, those snapshots are deleted, and we continue scanning the eventlist.

The entire DeltaGraph can thus be constructed in a single pass over E, assuming sufficient memory is available. At any point during the construction, we may have up to $k-1$ snapshots for each level of the DeltaGraph constructed so far. For higher values of k, this can lead to very high memory requirements. However, we use the GraphPool data structure to maintain these snapshots in an overlaid fashion to decrease the total memory consumption. We were able to scale to reasonably large graphs using this technique. Further scalability is achieved by making multiple passes over E, processing one partition in each pass (Section IV-B).

Updates to the Current graph: Ongoing updates to the network are recorded in a separate eventlist. After L such events are recorded, the eventlist is added to the DeltaGraph at the end, a new leaf-level snapshot is created, and the DeltaGraph hierarchy is adjusted to accommodate this new leaf node (by creating appropriate interior nodes). During the last phase, i.e., during the modification of its hierarchy, the DeltaGraph is locked and no reads are allowed.

G. Extensibility

To efficiently support specific types of queries or tasks, it is beneficial to maintain and index auxiliary information in the DeltaGraph, that can be used to effectively answer those queries. We allow extending the DeltaGraph functionality through user-defined modules and functions for this purpose. Due to space constraints, we sketch the key ideas briefly here; see [13] for further details. In essence, the user can supply functions that compute **auxiliary information** for each snapshot, that will be automatically indexed along with the original graph data. In addition, the user may also supply functions that operate on the auxiliary information deltas during retrieval, that can be used to directly answer specific types of queries. We illustrate the extensibility framework through a proof-of-concept example of a **subgraph pattern matching** index. Techniques for subgraph pattern matching are very well studied in literature (see, e.g., [28], [11]). We evaluated our implementation on Dataset 1 (details in the Section VII), and assigned labels to each node by randomly picking one from a list of ten labels. We built the index by indexing all paths of length 4 (see [13] for details). We were able to run a subgraph pattern query in 148 seconds to find all occurrences of a given pattern query, returning a total of 14109 matches over the entire history of the network.

V. DeltaGraph Analysis

Next we develop analytical models for storage space, memory consumption, and query latencies for a DeltaGraph.

A. Model of Graph Dynamics

Let G_0 denote the initial graph as of time 0, and let $G_{|E|}$ denote the graph after $|E|$ events. To develop the analytical

models, we make some simplifying assumptions, the most critical being that we assume a constant rate of inserts or deletes. Specifically, we assume that a δ_* fraction of the events result in an addition of an element to a graph (i.e., inserts), and ρ_* fraction of the events result in removal of an existing element from the graph (deletes). An update is captured as a delete followed by an insert. Thus, we have that $|G_{|E|}| = |G_0| + |E| \times \delta_* - |E| \times \rho_*$. We have that $\delta_* + \rho_* < 1$, but not necessarily $= 1$ because of transient events that don't affect the graph size. Typically we have that $\delta_* > \rho_*$. If $\rho_* = 0$, we call the graph a *growing-only* graph.

Note that, the above model does not require that the graph change at a constant rate *over time*. In fact, the above model (and the DeltaGraph structure) don't explicitly reason about time but rather only about the events. To reason about graph dynamics over time, we need a model that captures *event density*, i.e., number of events that take place over a period of time. Let $g(t)$ denote the total number of events that take place from time 0 to time t. For most real-world networks, we expect $g(t)$ to be a super-linear function of t, indicating that the rate of change over time itself increases over time.

B. Differential Functions

Recall that a differential function specifies how the snapshot corresponding to an interior node should be constructed from snapshots corresponding to its children. The simplest differential function is *intersection*. However, for most networks, intersection does not lead to desirable behavior. For a growing-only graph, intersection results in a left-skewed DeltaGraph, where the delta sizes are lower on the part corresponding to the older snapshots. In fact, the root is exactly G_0 for a strictly growing-only graph.

Table I shows several other differential functions with better and tunable behavior. Let p be an interior node with children a and b. Let $\Delta(a,p)$ and $\Delta(b,p)$ denote the corresponding deltas. Further, let $b = a + \delta_{ab} - \rho_{ab}$.

Skewed: For the two extreme cases, $r = 0$ and $r = 1$, we have that $f(a,b) = a$ and $f(a,b) = b$ respectively. By using an appropriate value of r, we can control the sizes of the two deltas. For example, for $r = 0.5$, we get $p = a + \frac{1}{2}\delta_{ab}$. Here $\frac{1}{2}\delta_{ab}$ means that we randomly choose half of the events that comprise δ_{ab} (by using a hash function that maps the events to 0 or 1). So $|\Delta(a,p)| = \frac{1}{2}|\delta_{ab}|$, and $|\Delta(b,p)| = \frac{1}{2}|\delta_{ab}| + |\rho_{ab}|$.

Balanced: This differential function, a special case of **mixed**, ensures that the delta sizes are balanced across a and b, i.e., $|\Delta(a,p)| = |\Delta(b,p)| = \frac{1}{2}|\delta_{ab}| + \frac{1}{2}|\rho_{ab}|$. Note that, here we make an assumption that $a + \frac{1}{2}\delta_{ab} - \frac{1}{2}\rho_{ab}$ is a valid operation. A problem may occur because an event $\in \rho_{ab}$ may be selected for removal, but may not exist in $a + \frac{1}{2}\delta_{ab}$. We can ensure that this does not happen by using the same hash function for choosing both $\frac{1}{2}\delta_{ab}$ and $\frac{1}{2}\rho_{ab}$.

Empty: This special case makes the DeltaGraph approach identical to the Copy+Log approach.

The other functions shown in Table I can be used to expose more subtle trade-offs, but our experience with these functions suggests that, in practice, Intersection, Union, and Mixed functions are likely to be sufficient for most situations.

TABLE I
DIFFERENTIAL FUNCTIONS

Name	Description
Intersection	$f(a,b,c\dots) = a \cap b \cap c \dots$
Union	$f(a,b,c\dots) = a \cup b \cup c \dots$
Skewed	$f(a,b) = a + r.(b-a)$, $0 \le r \le 1$
Right Skewed	$f(a,b) = a \cap b + r.(b - a \cap b)$, $0 \le r \le 1$
Left Skewed	$f(a,b) = a \cap b + r.(a - a \cap b)$, $0 \le r \le 1$
Mixed	$f(a,b,c\dots) = a + r_1.(\delta_{ab} + \delta_{bc}\dots) - r_2.(\rho_{ab} + \rho_{bc}\dots)$, $0 \le r_2 \le r_1 \le 1$
Balanced	$f(a,b,c\dots) = a + \frac{1}{2}.(\delta_{ab} + \delta_{bc}\dots) - \frac{1}{2}.(\rho_{ab} + \rho_{bc}\dots)$
Empty	$f(a,b,c\dots) = \emptyset$

C. Space and Time Estimation Models

Next, we develop analytical models for various quantities of interest in the DeltaGraph, including the space required to store it, the distribution of the delta sizes across levels, and the snapshot retrieval times. We focus on the Balanced and Intersection differential functions; we omit detailed derivations for lack of space, and refer the reader to [13] instead.

We make several simplifying assumptions in the analysis below. As discussed above, we assume constant rates of inserts and deletes. Let L denote the leaf-eventlist size, and let E denote the complete eventlist corresponding to the historical trace. Thus, we have $N = \frac{|E|}{L} + 1$ leaf nodes. We denote by k the arity of the graph, and assume that N is a power of k (resulting in a complete k-ary tree). We number the DeltaGraph levels from the bottom, starting with 1 (i.e., the bottommost level is called the first level).

Balanced Function: Although it appears somewhat complex, the Balanced differential function is the easiest to analyze. The total amount of space required to store all the deltas (excluding the one from the empty super-root node to the root) can be shown to be:

$$\frac{(\log_k N - 1)}{2}(k-1)(\delta_* + \rho_*)|E|$$

The size of the snapshot corresponding to the root itself can be seen to be: $|G_0| + \frac{1}{2}(\delta_* - \rho_*)|E|$ (independent of k). Although this may seem high, we note that the size of the current graph ($G_{|E|}$) is: $|G_0| + (\delta_* - \rho_*)|E|$, which is larger than the size of the root. Further, there is a significant overlap between the two, especially if $|G_0|$ is large, making it relatively cheap to materialize the root.

Also, because of symmetry, we can show that the total weight of the shortest path between the root and any leaf is: $\frac{1}{2}(\delta_* + \rho_*)|E|$, resulting in balanced query latencies for the snapshots (for specific timepoints corresponding to the same leaf-eventlist, there are small variations because of different portions of the leaf-eventlist that need to be processed).

Intersection: On the other hand, the Intersection function is much trickier to analyze. In fact, just calculating the size of the intersection for a sequence of snapshots is non-trivial in the general case. As above, consider a graph containing $|E|$ events. The root of the DeltaGraph contains all events that were not deleted from G_0 during that event trace. We state

the following analytical formulas for the size of the root for some special cases without full derivations.

$\rho_* = 0$: For a growing-only graph, $|root| = G_0$.

$\delta_* = \rho_*$: Here, the size of the graph remains constant (i.e., $G_{|E|} = G_0$). We can show that: $|root| = |G_0|e^{-\frac{|E|\delta_*}{|G_0|}}$.

$\delta_* = 2\rho_*$: $|root| = \frac{|G_0|^2}{|G_0|+\rho_*|E|}$.

The last two formulas both confirm our intuition that, as the total number of events increases, the size of the root goes to zero. Similar expressions can be derived for the sizes of any specific interior node or the deltas, by plugging in appropriate values of $|E|$ and $|G_0|$. We omit resulting expressions for the total size of the index for the latter two cases.

The Intersection function does have a highly desirable property that, the total weight of the shortest path between the super-root and any leaf is exactly the size of that leaf. Since an interior node contains a subset of the events in each of its children, we only need to fetch the remaining events to construct the child. However, this means that the query latencies are typically skewed, with the older snapshots requiring less time to construct than the newer snapshots (that are typically larger).

D. Discussion

We briefly discuss the impact of different construction parameters and suggest strategies for choosing the right parameters. We then briefly present a qualitative comparison with interval trees, segment trees, and the Copy+Log approach.

Effect of different construction parameters: The parameters involved in the construction of the DeltaGraph give it high flexibility, and must be chosen carefully. The optimal choice of the parameters is highly dependent on the application scenario and requirements. The effect of arity is easy to quantify in most cases: higher arity results in lower query access times, but usually much higher disk space utilization (even for the Balanced function, the query access time becomes dependent on k for a more realistic cost model where using a higher number of queries to fetch the same amount of information takes more time). Parameters such as r (for Skewed function) and r_1, r_2 (for Mixed function) can be used to control the query retrieval times over the span of the eventlist. For instance, if we expect a larger number of queries to be accessing newer snapshots, then we should choose higher values for these parameters.

The choice of differential function itself is quite critical. Intersection typically leads to lower disk space utilization, but also highly skewed query latencies that cannot be tuned except through memory materialization. Most other differential functions lead to higher disk utilization but provide better control over the query latencies. Thus if disk utilization is of paramount importance, then Intersection would be the preferred option, but otherwise, the Mixed function (with the values of r_1 and r_2 set according to the expected query workload) would be the recommended option.

Fine-tuning the values of these parameters also requires knowledge of $g(t)$, the event density over time. The analytical models that we have developed reason about the retrieval times for the leaf snapshots, but these must be weighted using $g(t)$ to reason about retrieval times over time. For example, the Balanced function does not lead to uniform query latencies over time for graphs that show super-linear growth. Instead, we must choose $r_1, r_2 > 0.5$ to guarantee uniform query latencies over time in that case.

Qualitative comparison with other approaches: The Copy+Log approach can be seen as a special case of DeltaGraph with Empty differential function (and arity = N). Compared to interval trees, DeltaGraph will almost always need more space, but its space consumption is usually lower than segment trees. Assume that $\delta_* + \rho_* = 1$ (this is the worst case for DeltaGraph). Then, for the Balanced function, with arity (k) = 2, the disk space required is $O(|E| \log N)$. Since the number of *intervals* is at least $|E|/2$, the space requirements for interval trees and segment trees are $O(|E|)$ and $O(|E| \log |E|)$ respectively. For growing-only graphs and the Intersection function, we see similar behavior. In most other scenarios, we expect the total space requirement for DeltaGraph to be somewhere in between $O(|E|)$ and $O(|E| \log N)$, and lower if $\delta_* + \rho_* \ll 1$ (which is often the case for social networks).

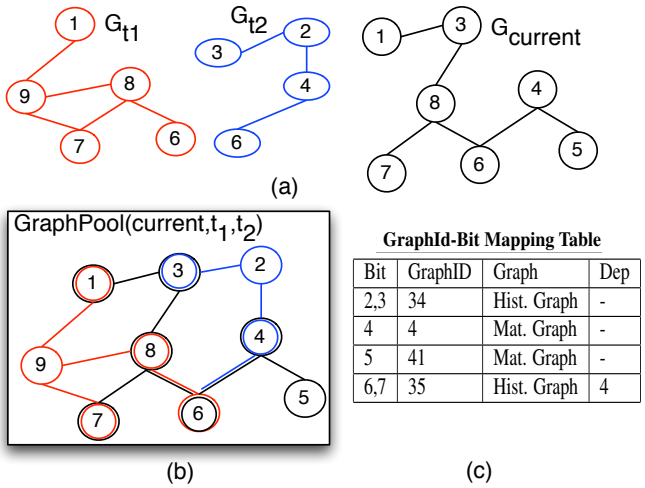
Regarding query latencies, for Intersection without any materialization, the amount of information retrieved for answering a snapshot query is exactly the size of the snapshot. Both interval trees and segment trees behave similarly. On the other hand, if the root or some of the higher levels of the DeltaGraph are materialized, then the query latencies could be significantly lower than what we can achieve with either of those approaches. For Balanced function, if the root is materialized, then the *average* query latencies are similar for the three approaches. However, for the Balanced function, the retrieval times do not depend on the size of the retrieved snapshot, unlike interval and segment trees, leading to more predictable and desirable behavior. Again, with materialization, the query latencies can be brought down even further.

VI. GraphPool

The in-memory graphs are stored in the in-memory *GraphPool* in an overlapping manner. In this section, we briefly describe the key ideas behind this data structure.

Description: GraphPool maintains a single graph that is the union of all the *active* graphs including: (1) the current graph, (2) historical snapshots, and (3) materialized graphs (Figure 5). Each component (node or edge), and for each attribute, each of its possible attribute values, are associated with a BitMap string (called BM henceforth), used to decide which of the active graphs contain that component or attribute. A *GraphID-Bit* mapping table is used to maintain the mapping of bits to different graphs. Figure 5(c) shows an example of such a mapping. Each historical graph that has been fetched is assigned two consecutive Bits, $\{2i, 2i+1\}, i \geq 1$. Materialized graphs, on the other hand, are only assigned one Bit.

Bits 0 and 1 are reserved for the current graph membership. Specifically, Bit 0 tells us whether the element belongs to the



GraphId-Bit Mapping Table

Bit	GraphID	Graph	Dep
2,3	34	Hist. Graph	-
4	4	Mat. Graph	-
5	41	Mat. Graph	-
6,7	35	Hist. Graph	4

Fig. 5. (a) Graphs at times $t_{current}$, t_1, and t_2; (b) GraphPool consisting of overlaid graphs; (c) GraphID-Bit Mapping Table

current graph or not. Bit 1, on the other hand, is used for elements that may have been recently deleted, but are not part of the DeltaGraph index yet. A Bit associated with a materialized graph is interpreted in a straightforward manner.

Using a single bit for a historical graph misses out on a significant optimization opportunity. Even if a historical graph differs from the current graph or one of the materialized graphs in only a few elements, we would still have to set the corresponding bit appropriately for all the elements in the graph. We can use the bit pair, $\{2i, 2i+1\}$, to eliminate this step. We mark the historical graph as being dependent on a materialized graph (or the current graph) in such a case. For example, in Figure 5(c), historical snapshot 35 is dependent on materialized graph 4. If Bit $2i$ is true, then the membership of an element in the historical graph is identical to its membership in the materialized graph (i.e., if present in one, then present in another). On the other hand, if Bit $2i$ is false, then Bit $2i+1$ tells us whether the element is contained in the historical graph or not (independent of the materialized graph).

When a graph is pulled into the memory either in response to a query or for materialization, it is overlaid onto the current in-memory graph, edge by edge and node by node. The number of graphs that can be overlaid simultaneously depends primarily on the amount of memory required to contain the union of all the graphs. The BM size is dynamically adjusted to accommodate more graphs if needed, and overall does not occupy significant space. The determination of whether to store a graph as being dependent on a materialized graph is made at the query time. During the query plan construction, we count the total number of events that need to be applied to the materialized graph, and if it is small relative to the size of the graph, then the fetched graph is marked as being dependent on the materialized graph.

Clean-up of a graph from memory: When a historical graph is no longer needed, it needs to be *cleaned*. Cleaning up a graph is logically a reverse process of fetching it into the memory. The naive way would be to go through all the elements in the graph, and reset the appropriate bit(s), and delete the element if no bits are set. However the cost of doing this can be quite high. We instead perform clean-up in a lazy fashion, periodically scanning the GraphPool in the absence of query load, to reset the bits, and to see if any elements should be deleted. Also, in case the system is running low on memory and there are one or more unneeded graphs, the Cleaner thread is invoked and not interrupted until the desired amount of memory is liberated.

VII. Empirical Evaluation

We present the results of a comprehensive experimental evaluation conducted to evaluate the performance of our prototype system, implemented in Java using the Kyoto Cabinet key-value store as the underlying persistent storage. The system provides a programmatic API including the API discussed in Section III-B1; in addition, we have implemented a Pregel-like iterative framework for distributed processing, and the subgraph pattern matching index presented in Section IV-G.

Datasets: We used three datasets in our experimental study.

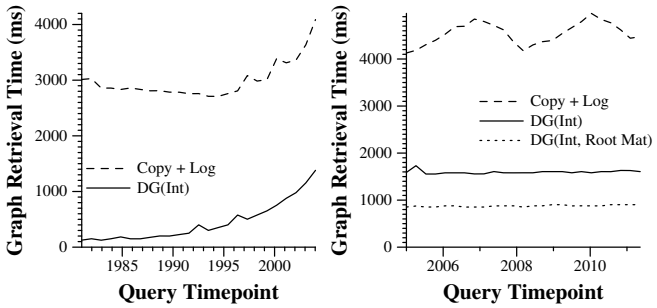
(1) Dataset 1 is a growing-only co-authorship network extracted from the DBLP dataset, with $2M$ edges in all. The network starts empty and grows over a period of seven decades. The nodes (authors) and edges (co-author relationships) are added to the network, and no nodes or edges are dropped. At the end, the total number of unique nodes present in the graph is around 330,000, and the number of edges with unique end points is $1.04M$. Each node was assigned 10 attribute key-value pairs, generated at random.

(2) Dataset 2 is a randomly generated historical trace with the final graph of Dataset 1 as the starting snapshot, followed by $2M$ events where $1M$ edges added and $1M$ edges are deleted over time.

(3) Dataset 3 is a randomly generated historical trace with a starting snapshot containing 10 million ($10M$) edges and $3M$ nodes (from a *patent citation network*), followed by $100M$ events, $50M$ edge additions and $50M$ edge deletions.

Experimental Setup: We created a partitioned index for Dataset 3 and deployed a parallel framework for PageRank computation using 7 machines, each with a single Amazon EC2 core and approximately 1.4GB of available memory. Each DeltaGraph partition was approximately 2.2GB. Note that the index is stored in a compressed fashion (using built-in compression in Kyoto Cabinet). On average, it took us **23.8 seconds to calculate PageRank** for a specific graph snapshot, including the snapshot retrieval time. This experiment illustrates the effectiveness of our framework at scalably handling large historical graphs.

For the rest of the experimental study, we report results for Datasets 1 and 2; the techniques we compare against are centralized, and further the cost of constructing the index makes it hard to run experiments that evaluate the effect of the construction parameters. Unless otherwise specified, we used a single Amazon EC2 core (with 1.4GB memory).



(a) Performance: Dataset 1 (b) Performance: Dataset 2

Fig. 6. Comparing DeltaGraph and Copy+Log. Int and Bal denote the Intersection and Balanced functions respectively.

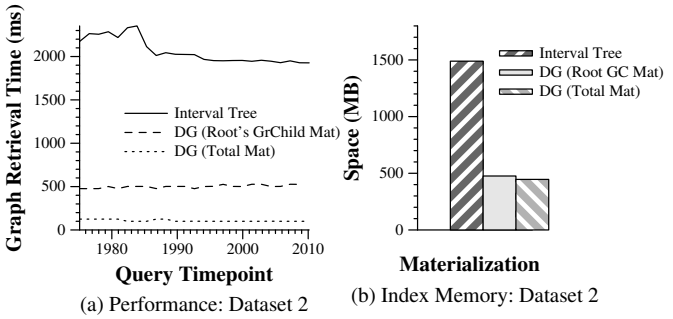
Comparison with other storage approaches: We begin with comparing our approach with **in-memory interval trees**, and Copy+Log approach. Both of those were integrated into our system such that any of the approaches could be used to fetch the historical snapshots into the GraphPool, and we report the time taken to do so.

Figure 6 shows the results of our comparison between Copy+Log and DeltaGraph approaches for time taken to execute 25 uniformly spaced queries on Datasets 1 and 2. The leaf-eventlist sizes were chosen so that the disk storage space consumed by both the approaches was about the same. For similar disk space constraints (450MB and 950MB for Dataset 1 and 2, respectively), the DeltaGraph could afford a smaller size of L and hence higher number of snapshots than the Copy+Log approach. As we can see, the best DeltaGraph variation outperformed the Copy+Log approach by a factor of at least 4, and orders of magnitude in several cases.

Figure 7 compares an in-memory interval tree and two DeltaGraph variations: (1) with low materialization, (2) with all leaf nodes materialized. We compared these configurations for time taken to execute 25 queries on Dataset 2, using $k = 4$ and $L = 30000$. We can see that both the DeltaGraph approaches outperform the interval tree approach, while using significantly less memory than the interval tree (even with total materialization). The largely disk-resident DeltaGraph with root's grandchildren materialized is more than four times as fast as the regular approach, whereas the total materialization approach, a more fair comparison, is much faster.

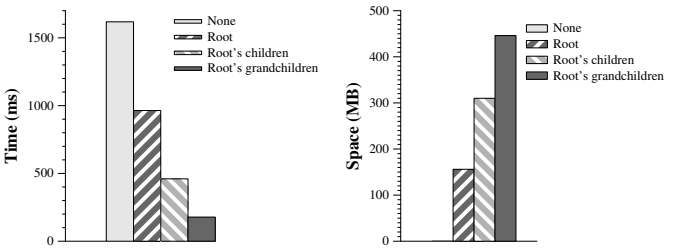
We also evaluated a naive approach similar to the Log technique, with raw events being read from input files directly (not shown in the above plots). The average retrieval times were worse than DeltaGraph by factors of 20 and 23 for Datasets 1 and 2 respectively.

Materialization: Figure 8 shows the benefits of materialization for a DeltaGraph and the associated cost in terms of memory, for Dataset 2 with arity = 4 and using the Intersection differential function. We compared four different situations: (a) no materialization, (b) root materialized, (c) both children of the root materialized, and (d) all four grandchildren of the root materialized. The results are as expected – we can significantly reduce the query latencies (up to a factor of 8) at the expense



(a) Performance: Dataset 2 (b) Index Memory: Dataset 2

Fig. 7. Performance of different DeltaGraph configurations vs. Interval Tree



(a) Average Query Time (b) Materialization Memory Requirement

Fig. 8. Effect of materialization

of higher memory consumption.

GraphPool memory consumption: Figure 9(a) shows the total (cumulative) memory consumption of the GraphPool when a sequence of 100 singlepoint snapshot retrieval queries, uniformly spaced over the life span of the network, is executed against Datasets 1 and 2. By exploiting the overlap between these snapshots, the GraphPool is able to maintain a large number of snapshots in memory. For Dataset 2, if the 100 graphs were to be stored disjointly, the total memory consumed would be 50GB, whereas the GraphPool only requires about 600MB. The plot of Dataset 1 is almost a constant because, for this dataset, any historical snapshot is a subset of the current graph. The minor increase toward the end is due to the increase in the BM size, required to accommodate new queries.

Multicore Parallelism: Figure 9(b) shows the advantage of concurrent query processing on a multi-core processor using a partitioned DeltaGraph approach, where we retrieve the graph parallely using multiple threads. We observe near-linear speedups further validating our parallel design.

Multipoint queries: Figure 9(c) shows the time taken to retrieve multiple graphs using our multipoint query retrieval algorithm, and multiple invocations of the single query retrieval algorithm on Dataset 1. The x-axis represents the number of snapshot queries, which were chosen to be 1 month apart. As we can see, the advantages of multipoint query retrieval are quite significant because of the high overlap between the retrieved snapshots.

Advantages of columnar storage: Figure 9(d) shows the performance benefits of our columnar storage approach for Dataset 1. As we can see, if we are only interested in the network structure, our approach can improve query latencies

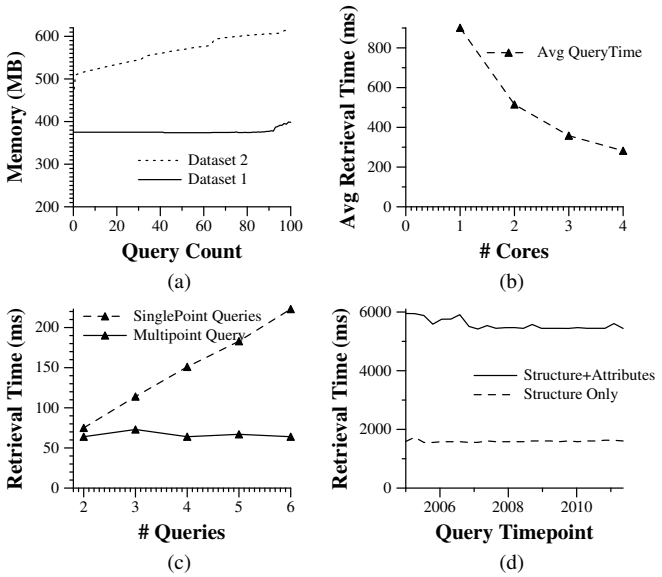


Fig. 9. (a) Cumulative GraphPool memory consumption; (b) Multi-core parallelism (Dataset 2); (c) Multipoint query execution vs multiple singlepoint queries; (d) Retrieval with and without attributes (Dataset 2)

by more than a factor of 3.

BM penalty: We compared the penalty of using the BM filtering procedure in GraphPool, by doing a PageRank computation without and with use of BMs. We observed that the execution time increases by 7%, from 1890ms to 2014ms.

Additional experiments: We have also run additional experiments that illustrate the **effects of different DeltaGraph construction parameters** like the arity (k), leaf-eventlist sizes (L), and differential functions (f). Due to space constraints, we refer the reader to the extended version of the paper [13].

VIII. Conclusions and Future Work

In this paper, we presented techniques for managing historical data for large information networks, and for executing snapshot retrieval queries on them. We presented DeltaGraph, a distributed hierarchical structure that enables compact storage of the historical trace of a network, and GraphPool, an in-memory data structure that allows us to maintain and operate upon a large number of snapshots simultaneously. Our experimental evaluation shows that the choice of DeltaGraph is superior to the existing alternatives. We showed both analytically and empirically that the flexibility and tunability of DeltaGraph helps control the distribution of query access times through appropriate parameter choices at construction time, and memory materialization at runtime. Our experimental evaluation demonstrated the impact of many of our optimizations, including multi-query optimization and columnar storage. Our work so far has also opened up many opportunities for further work, including developing techniques for processing different types of temporal queries over the historical trace, that we are planning to pursue in future work. We are also studying how sketch-based techniques [2] may be used for more efficient, albeit approximate, query processing over historical graphs.

Acknowledgment

This work was supported in part by the Air Force Research Lab (AFRL) under contract FA8750-10-C-0191, an IBM Collaborative Research Award, and an Amazon AWS in Education Research grant.

References

[1] J. Ahn, C. Plaisant, and B. Shneiderman. A task taxonomy of network evolution analysis. *HCIL Tech. Reports*, April 2011.
[2] K. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *PODS*, pages 5–14, 2012.
[3] L. Arge and J. Vitter. Optimal dynamic interval management in external memory. In *FOCS*, 1996.
[4] S. Asur, S. Parthasarathy, D. Ucar. An event-based framework for characterizing the evolutionary behavior of interaction graphs. *TKDD*, 2009.
[5] T. Berger-Wolf and J. Saia. A framework for analysis of dynamic social networks. In *KDD*, 2006.
[6] G. Blankenagel, R. Guting. External segment trees. *Algorithmica*, 1994.
[7] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. *ACM TODS*, 29(1):2–42, 2004.
[8] C. Date, H. Darwen, and N. Lorentzos. *Temporal data and the relational model*. Elsevier, 2002.
[9] S. Ghandeharizadeh, R. Hull, D. Jacobs. Heraclitus: elevating deltas to be first-class citizens in a database programming language. *ACM TODS*, 21(3), 1996.
[10] D. Greene, D. Doyle, and P. Cunningham. Tracking the evolution of communities in dynamic social networks. In *ASONAM*, 2010.
[11] H. He and A. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.
[12] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, 2009.
[13] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. *CoRR*, abs/1207.5777, 2012.
[14] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.
[15] N. Lam and R. Wong. A fast index for XML document version management. In *APWeb*, 2003.
[16] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM TKDD*, 2007.
[17] D. Lomet, M. Hong, R. Nehme, and R. Zhang. Transaction time indexing with version compression. In *VLDB*, 2008.
[18] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *PODC*, 2009.
[19] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *VLDB*, 2001.
[20] I. McCulloh and K. Carley. Social network change detection. *Center for the Computational Analysis*, 2008.
[21] S. Navlakha and C. Kingsford. Network archaeology: Uncovering ancient networks from present-day interactions. *PLoS Comp Bio*, 2011.
[22] G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: a survey. *IEEE TKDE*, 7(4):513 –532, aug 1995.
[23] A. Pugliese, O. Udrea, and V. Subrahmanian. Scaling RDF with time. In *WWW*, 2008.
[24] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historial evolving graph sequences. In *VLDB*, 2011.
[25] B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2), 1999.
[26] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *ICDE*, 2012.
[27] B. Shao, H. Wang, and Y. Li. The Trinity graph engine. Technical Report MSR-TR-2012-30, Microsoft Research, 2012.
[28] D. Shasha, J. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.
[29] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.
[30] L. Tang, H. Liu, J. Zhang, and Z. Nazeri. Community evolution in dynamic multi-mode networks. In *KDD*, 2008.
[31] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (ed.). *Temporal Databases: Theory, Design, and Implementation*. 1993.
[32] J. Tappolet and A. Bernstein. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *ESWC*, pages 308–322, 2009.
[33] V. Tsotras and N. Kangelaris. The Snapshot Index: an I/O- optimal access method for timeslice queries. *Inf. Syst.*, 1995.