Storing and Analyzing Historical Graph Data at Scale

Udayan Khurana
IBM TJ Watson Research Center
ukhurana@us.ibm.com

Amol Deshpande
University of Maryland
amol@cs.umd.edu

ABSTRACT

The work on large-scale graph analytics to date has largely focused on the study of static properties of graph snapshots. However, a static view of interactions between entities is often an oversimplification of several complex phenomena like the *spread of epidemics*, *information diffusion*, *formation of online communities*, and so on. Being able to find temporal interaction patterns, visualize the evolution of graph properties, or even simply compare snapshots across time, adds significant value in reasoning over graphs. However, due to the lack of underlying data management support, an analyst today has to manually navigate the added temporal complexity of dealing with large evolving graphs. In this paper, we present a system, called *Historical Graph Store*, that enables users to store large volumes of historical graph data and to express and run complex temporal graph analytical tasks against that data. It consists of two key components: (1) a *Temporal Graph Index* (TGI), that compactly stores large volumes of historical graph evolution data in a partitioned and distributed fashion – TGI also provides support for retrieving snapshots of the graph as of any timepoint in the past or evolution histories of individual nodes or neighborhoods; and (2) a *Temporal Graph Analysis Framework* (TAF), for expressing complex temporal analytical tasks and for executing them in an efficient and scalable manner using Apache Spark. Our experiments demonstrate our system's efficient storage, retrieval and analytics across a wide variety of queries on large volumes of historical graph data.

1. INTRODUCTION

Graphs are useful in capturing behavior involving interactions between entities. Several processes are naturally represented as graphs – social interactions between people, financial transactions, biological interactions among proteins, geospatial proximity of infected livestock, and so on. Many problems based on such graph models can be solved using well-studied algorithms from graph theory or network science. Examples include finding driving routes by computing shortest paths on a network of roads, finding user communities through dense subgraph identification in a social network, and many others. Numerous graph data management systems have been developed over the last decade, including specialized graph database systems like Neo4j, Titan, etc., and large-scale graph processing frameworks such as GraphLab [27], Pregel [29], Giraph, GraphX [12], GraphChi [24], etc.

However much of the work to date, especially on cloud-scale graph data management systems, focuses on managing and analyzing a single (typically, current) static snapshot of the data. In the real world, however, interactions are a dynamic affair and any graph that abstracts a real-world process changes over time. For instance, in online social media, the friendship network on Facebook or the "follows" network on Twitter change steadily over time, whereas the "mentions" or the "retweet" networks change much more rapidly. Dynamic cellular networks in biology, evolving citation networks in publications, dynamic financial transactional networks, are few other examples of such data. Lately, we have seen an increasing merit in dynamic modeling and analysis of network data to obtain crucial insights in several domains such as cancer prediction [38], epidemiology [15], organizational sociology [16], molecular biology [9], information spread on social networks [26] amongst others.

In this work, our focus is on providing the ability to analyze and to reason over the entire history of the changes to a graph. There are many different types of analyses of interest. For example, an analyst may wish to study the evolution of well-studied static graph properties such as centrality measures, density, conductance, etc., over time. Another approach is through the search and discovery of temporal patterns, where the events that constitute the pattern are spread out over time. Comparative analysis, such as juxtaposition of a statistic over time, or perhaps, computing aggregates such as *max* or *mean* over time, possibly gives another style of knowledge discovery into temporal graphs. Most of all, a primitive notion of just being able to access past states of the graphs and performing simple static graph analysis, empowers a data scientist with the capacity to perform analysis in arbitrary and unconventional patterns.

Supporting such a diverse set of temporal analytics and querying over large volumes of historical graph data requires addressing several data management challenges. Specifically, there is a want of techniques for storing the historical information in a compact manner, while allowing a user to retrieve graph snapshots as of any time point in the past or the evolution history of a specific node or a specific neighborhood. Further, the data must be stored and queried in a distributed fashion to handle the increasing scale of the data. There is also a need for an expressive, high-level, easy-to-use programming framework that will allow users to specify complex temporal graph analysis tasks, while ensuring those tasks can be executed efficiently in a data-parallel fashion across a cluster.

In this paper, we present a graph data management system, called *Historical Graph Store (HGS)*, that provides an ecosystem for managing and analyzing large historical traces of graphs. HGS con-

sists of two key distinct components. First, the *Temporal Graph Index (TGI)*, is an index that compactly stores the entire history of a graph by appropriately partitioning and encoding the differences over time (called *deltas*). These deltas are organized to optimize the retrieval of several temporal graph primitives such as neighborhood versions, node histories, and graph snapshots. TGI is designed to use a distributed key-value store to store the partitioned deltas, and can thus leverage the scalability afforded by those systems (our implementation uses Apache Cassandra[1] key-value store). TGI is a tunable index structure, and we investigate the impact of tuning the different parameters through an extensive empirical evaluation. TGI builds upon our prior work on DeltaGraph [21], where the focus was on retrieving individual snapshots efficiently; TGI extends DeltaGraph to support efficient retrieval of subgraphs instead of only full snapshots, retrieval of histories of nodes or subgraphs over past time intervals, and features a highly scalable design over DeltaGraph.

The second component of HGS is a *Temporal Graph Analysis Framework (TAF)*, which provides an expressive framework to specify a wide range of temporal graph analysis tasks. TAF is based on a novel set of *temporal graph operands* and *operators* that enable parallel execution of the specified tasks at scale in a cluster environment. The execution engine is implemented on Apache Spark [40], a large-scale in-memory cluster computing framework.

Outline: The rest of the paper is organized as follows. In Section 2, we survey the related work on graph data stores, temporal indexing, and other topics relevant to the scope of the paper. In Section 3, we provide a sketch of the overall system, including key aspects of the underlying components. We then present TGI and TAF in detail in Sections 4 and 5, respectively. In Section 6, we provide an empirical evaluation, and and conclude with a summary and a list of future directions in Section 7.

2. RELATED WORK

In the recent years, there has been much work on graph storage and graph processing systems and numerous systems have been designed to address various aspects of graph data management. Some examples include Neo4J, Titan[2], GBase [19], Pregel [29], Giraph, GraphX [12], GraphLab [27], and Trinity [36]. These systems use a variety of different models for representation, storage, and querying, and there is a lack of standardized or widely accepted models for the same. Most graph querying happens through programmatic access to graphs in languages such as Java, Python or C++. Graph libraries such as Blueprints[3] provide a rich set of implementations for graph theoretic algorithms. SPARQL [33] is a language used to search patterns in linked data. It works on an underlying RDF representation of graphs. T-SPARQL [13] is a temporal extension of SPARQL. He et al. [17], provide a language for finding subgraph patterns using a graph as a query primitive. Gremlin[4] is a graph traversal language over the property graph data model, and has been adopted by several open-source systems. For large-scale graph analysis, perhaps the most popular framework is the vertex-centric programming framework, adopted by Giraph, GraphLab, GraphX, and several other systems; there have also been several proposals for richer and more expressive programming frameworks in recent years. However, most of these prior systems largely focus on analyzing a single snapshot of the graph data, with very little support for handling dynamic graphs, if any.

[1]https://cassandra.apache.org
[2]http://thinkaurelius.github.io/titan/
[3]https://github.com/tinkerpop/blueprints/wiki
[4]https://github.com/tinkerpop/gremlin

A few recent papers address the issues of storage and retrieval in dynamic graphs. In our prior work, we proposed DeltaGraph [21], an index data structure that compactly stores the history of all changes in a dynamic graph and provides efficient snapshot reconstruction. G* [25] stores multiple snapshots compactly by utilizing commonalities. ImmortalGraph [30] is an in-memory system for processing dynamic graphs, with the objectives of shared storage and computation for overlapping snapshots. Ghrab et al. [11] provide a system of network analytics through labeling graph components. Gedik et al. [10], describe a block-oriented and cache-enabled system to exploit spatio-temporal locality for solving temporal neighborhood queries. Koloniari et al. [23] also utilize caching to fetch selective portions of temporal graphs they refer to as partial views. LLAMA [28] uses multiversioned arrays to represent a mutating graph, but their focus is primarily on in-memory representation. There is also recent work on streaming analytics over dynamic graph data [8, 7], but it typically focuses on analyzing only the recent activity in the network (typically over a sliding window).

Temporal graph analytics is an area of growing interest. Evolution of shortest paths in dynamic graphs has been studied by Huo et al. [18], and Ren et al. [34]. Evolution of community structures in graphs has been of interest as well [5, 14]. Change in page rank with evolving graphs [3], and the study of change in centrality of vertices, path lengths of vertex pairs, etc. [32], also lie under the larger umbrella of temporal graph analysis. Ahn et al. [1] provide a taxonomy of analytical tasks over evolving graphs. Barrat et al. [4], provide a good reference for studying several dynamic processes modeled over graphs. Our system significantly reduces the effort involved in building and deploying such analytics over large volumes of graph data.

Temporal data management for relational databases was a topic of active research in the 80s and early 90s. Snapshot index [39] is an I/O optimal solution to the problem of snapshot retrieval for transaction-time databases. Salzberg and Tsotras [35] present a comprehensive survey of temporal data indexing techinques, and discuss two extreme approaches to supporting snapshot retrieval queries, referred to as the *Copy* and *Log* approaches. While the copy approach relies on storing new copies of a snapshot upon every point of change in the database, the log approach relies on storing everything through changes. Their hybrid is often referred to as the *Copy+Log* approach. We omit a detailed discussion of the work on temporal databases, and refer the interested reader to a representative set of references [37, 31, 35].Other data structures, such as Interval Trees [2] and Segment trees [6] can also be used for storing temporal information. Temporal aggregation in scientific array databases is another related topic of interest, but the challenges there are significantly different. Kaufmann et al. [20] propose an in-memory index in SAP HANA that addresses temporal aggregation, joins, and snapshot construction. The applicability of temporal relational data management techniques to graphs is restricted due to lack of (efficient) support for graph specific retrieval such as fetching neighborhoods, or histories of nodes over time. Our work in this paper focuses on techniques for a wide variety of temporal graph retrieval and analysis on entire graph histories that are primarily stored on disk.

3. OVERVIEW

In this section, we introduce key aspects related to HGS. We begin with the data model, followed by the key challenges and concluding with an overview of the system.

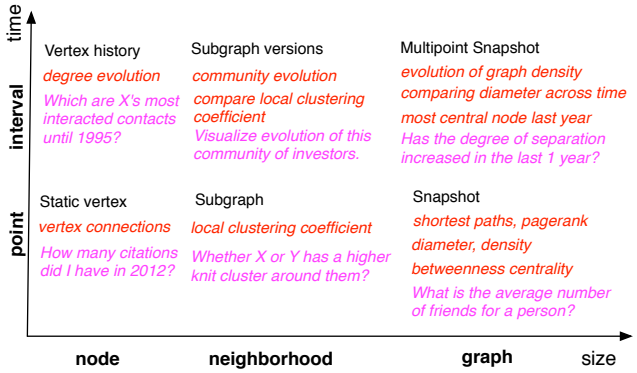


Figure 1: The scope of temporal graph analytics can be represented across two different dimensions - time and entity. The chart lists retrieval tasks (black), graph operations (red), example queries (magenta) at different granularities of time and entity size.

3.1 Data Model

Under a discreet notion of time, a time-evolving graph $G^T = (V^T, E^T)$ may be expressed as a collection of graph *snapshots* over different time points, $\{G^0 = (V^0, E^0), G^1, \ldots, G^t\}$. The vertex set V^i for a snapshot consists of a set of vertices (nodes), each of which has a unique identifier (constant over time), and an arbitrary number of key-value attribute pairs. The edge sets E^i consist of edges that each contain references to two valid nodes in the corresponding vertex set V^i, information about the direction of the edge, and an arbitrary list of key-value attribute pairs. A temporal graph can also be equivalently described by a set of changes to the graph over time. We call an atomic change at a specific timepoint in the graph an *event*. The changes could be structural, such as the addition or the deletion of nodes or edges, or be related to attributes such as an addition or a deletion or a change in the value of a node or an edge attribute. For instance, a new user joining the Facebook social network corresponds to an event of node creation; connecting to another user is an event of edge creation; changing location or posting an update are events of change and creation of attribute values, respectively. These approaches specified here as well as certain hybrids have been used in the past for the physical and logical modeling of temporal data. Our approach to temporal processing in this paper is best described using a *node-centric* logical model, i.e., the historical graph is seen as a collection of evolving vertices over time; the edges are considered as attributes of the nodes. This abstraction helps in our design of distributed storage of the graph and parallel execution of the analytical tasks.

3.2 Challenges

The nature of data management tasks in historical graph analytics can be categorized based on the scope of analysis using the dual dimensions of *time* and *entity* as illustrated with examples in Figure 1. The temporal scope of an analysis task can range from a single point in time to a long interval; the entity scope can range from a single node to the entire graph. While the diversity of analytical tasks provides a potential for a rich set of insights from historical graphs, it also poses several challenges in constructing a system that can perform those tasks. To the best of our knowledge, none of the existing systems address a majority of those challenges that are described below:

Compact storage with fast access: An natural tradeoff between index size and access latencies can be seen in the Log and Copy approaches for snapshot retrieval. Log requires minimal information to encode the graph's history, but incurs large reconstruction costs. Copy, on the other hand, provides direct access, but at the cost of excessive storage. The desirable index should consume space of the order of Log index but provide near direct access like Copy.

Time-centric versus entity-centric indexing: For *point* access such as past snapshot retrieval, a time-centric indexing such as DeltaGraph or Copy+Log is suitable. However, for version retrieval tasks such as retrieving a *node's history*, entity-centric indexing is the correct choice. Neither of the indexing approaches, however, are feasible in the opposite scenarios. Given the diversity of access needs, we require an index that works well with both styles of lookup at the same time.

Optimal granularity of storage for different queries: Query latencies for a graph also depend on the size of chunks in which the data is indexed. While larger granularities of storage incur wasteful data read for "node retrieval", a finely chunked graph storage would mean higher number of lookups and aggregation for a 2-hop neighborhood lookup. The physical and logical arrangement of data should take care of access needs at all granularities.

Coping with changing topology in a dynamic graph: It is evident that graph partitioning is inevitable in the storage and processing of large graphs. However, finding the appropriate strategy to maintain workable partitioning on a constantly *changing* graph is another challenge while designing a historical graph index.

Systematically expressing temporal graph analytics: A platform for expressing a wide variety of historical graph analytics requires an appropriate amalgam of temporal logic and graph theory. Additionally, utilizing a vast body of existing tools in network science is an engineering challenge and opportunity.

Appropriate abstractions for distributed, scalable analytics: Parallelization is key to scale up analytics for large graph datasets. It is essential that the underlying data-representations and operators in the analytical platform be designed for parallel computing.

3.3 System Overview

Figure 2 shows the architecture of our proposed Historical Graph Store. It consists of two main components:

Temporal Graph Index (TGI) records the entire history of a graph compactly while enabling efficient retrieval of several temporal graph primitives. It encodes various forms of differences (called *deltas*) in the graph, such as atomic events, changes in subgraphs over intervals of time, etc. It uses specific choices of graph partitioning, data replication, temporal compression and data placement to optimize the graph retrieval performance. TGI uses Cassandra, a distributed key-value store for the deltas. In Section 4, we describe the design details of TGI and the access algorithms.

Temporal Graph Analytics Framework (TAF) provides a *temporal node*-centric abstraction for specifying and executing complex temporal network analysis tasks. It helps the user analyze the history of the graph by means of simple yet expressive *temporal operators*. The abstraction of temporal graph through a *set of (temporal) nodes (SoN)* allows the framework to achieve computational scalability through distribution of tasks by node and time. TAF is built on top of Apache Spark to utilize its support for scalable, in-memory, cluster computation; TAF provides an option to utilize GraphX for static graph computation. We provide a Java and Python based library to specify the retrieval, computation and analysis tasks. In Section 5, we describe the details of the data and computational models, query processing, parallel data fetch aspects of the system, the analytical library along with a few examples.

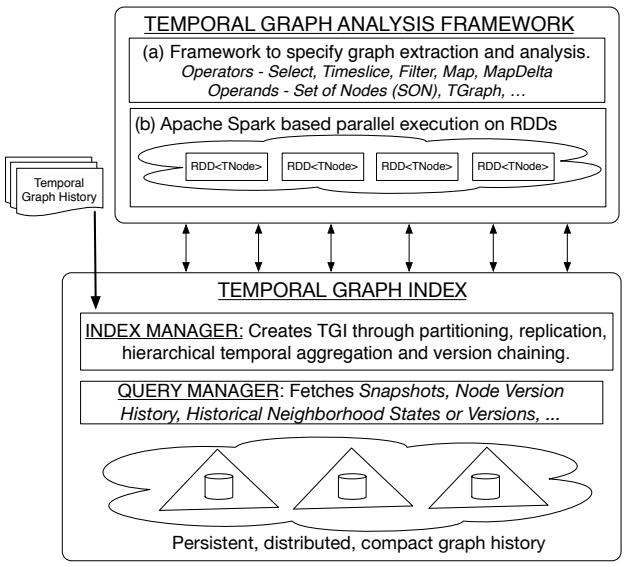


Figure 2: System Overview

4. TEMPORAL GRAPH INDEX

In this section, we investigate the issue of indexing temporal graphs. First, we introduce a *delta framework* to define any temporal index as a set of different changes or *deltas*. Using this framework, we are able to qualitatively compare the access costs and sizes of different alternatives for temporal graph indexing, including our proposed approach. We then present the Temporal Graph Index (TGI), that stores the entire history of a large evolving network in the cloud, and facilitates efficient parallel reconstruction for different graph primitives. TGI is a generalization of both entity and time-based indexing approaches and can be tuned to suit specific workload needs. We claim that TGI is the minimal index that provides efficient access to a variety of primitives on a historical graph, ranging from past snapshots to versions of a node or neighborhood. We also describe the key partitioning strategies instrumental in scaling to large datasets across a cloud storage.

4.1 Preliminaries

We start with a few preliminary definitions that help us formalize the notion of the delta framework.

DEFINITION 1 (STATIC NODE). *A static node refers to the state of a vertex in a network at a specific time, and is defined as a set containing: (a) node-id, denoted I (an integer), (b) an edge-list, denoted E (captured as a set of node-ids), (c) attributes, denoted A, a map of key-value pairs.*

A *static edge* is defined analogously, and contains the node-ids for the two endpoints and the edge direction in addition to a map of key-value pairs. Finally, a *static graph component* refers to either a static edge or a static node.

DEFINITION 2 (DELTA). *A Delta (Δ) refers to either: (a) a static graph component (including the empty set), or (b) a difference, sum, union or intersection of two deltas.*

Such a definition of delta helps express the change in a wider context than merely difference of graph states at two points. It helps us articulate several temporal graph indexes including TGI and DeltaGraph in a single framework.

DEFINITION 3 (CARDINALITY AND SIZE). *The cardinality and the size of a delta are the unique and total number of static node or edge descriptions within it, respectively.*

DEFINITION 4 (Δ SUM). *A sum (+) over two deltas, Δ_1 and Δ_2, i.e., $\Delta_s = \Delta_1 + \Delta_2$ is defined over graph components in the two deltas as follows: (1) $\forall gc_1 \in \Delta_1$, if $\exists gc_2 \in \Delta_2$ s.t. $gc_1.I = gc_2.I$, then we add gc_2 to Δ_s, (2) $\forall gc_1 \in \Delta_1$ s.t. $\nexists gc_2 \in \Delta_2$ s.t. $gc_1.I = gc_2.I$, we add gc_1 to Δ_s, and (3) analogously the components present only in Δ_2 are added to Δ_s.*

Note that: $\Delta_1 + \Delta_2 = \Delta_2 + \Delta_1$ is not necessarily true due the order of changes. We also note that: $\Delta_1 + \emptyset = \Delta_1$, and $(\Delta_1 + \Delta_2) + \Delta_3 = \Delta_1 + (\Delta_2 + \Delta_3)$. Analogously, difference(-) is defined as a set difference over different components of the two deltas. $\Delta_1 - \phi = \Delta_1$ and $\Delta_1 - \Delta_1 = \phi$, are true, while, $\Delta_1 - \Delta_2 = \Delta_2 - \Delta_1$, does not necessarily hold.

DEFINITION 5 (Δ INTERSECTION). *An intersection of two deltas is defined as a set intersection over the the components of two deltas. $\Delta_1 \cap \phi = \phi$, is true for any delta. Similarly, union of two deltas $\Delta_\cup = \Delta_1 \cup \Delta_2$, consists of all elements from Δ_1 and Δ_2. The following is true for any delta: $\Delta_1 \cup \phi = \Delta_1$.*

Next we discuss and define some specific types of deltas:

DEFINITION 6 (EVENT). *An event is the smallest change that happens to a graph, i.e., addition or deletion of a node or an edge, or a change in an attribute value. An event is described around one time point. As a delta, an event concerning a graph component c, at time point t_e, is defined as the difference of state of c at and before t_e, i.e., $\Delta_{event}(c,t_e) = c(t_e) - c(t_e - 1)$.*

DEFINITION 7 (EVENTLIST). *An eventlist delta is a chronologically sorted set of event deltas. An eventlist's scope may be defined by the time duration, $(t_s,t_e]$, during which it defines all the changes that happened to the graph.*

DEFINITION 8 (EVENTLIST PARTITION). *An eventlist partition delta is is a chronologically sorted set of event deltas pertaining to a set of nodes, P, over a given time duration, $(t_s,t_e]$.*

DEFINITION 9 (SNAPSHOT). *A snapshot, G^{t_a} is the state of a graph G at a time point t_a. As a delta, it is defined as the difference of the state of the graph at t_a from an empty set, $\Delta_{snapshot}(G,t_a) = G(t_a) - G(-\infty)$.*

DEFINITION 10 (SNAPSHOT PARTITION). *A snapshot partition is a subset of a snapshot. It is identified by a subset P of all nodes in graph, G at time, t_a. It consists of all nodes in G at t_a and all the edges whose at least one end-point lies in P at time, t_a.*

4.2 Prior Techniques

The prior techniques for temporal graph indexing use changes or differences in various forms to encode time-evolving datasets. We can express them in the Δ-framework as follows. The ***Log*** index is equivalent to a set of all *event* deltas (equivalently, a single *eventlist* delta encompassing the entire history). The ***Copy+Log*** index can be represented as combination of: (a) a finite number of distinct *snapshot* deltas, and (b) *eventlist* deltas to capture the change between successive snapshots. Although we are not aware of a specific proposal for a ***vertex-centric*** index, however, a natural approach would be to maintain a set of *eventlist partition* deltas,

one for each node (with edge information replicated with the endpoints). The ***DeltaGraph*** index, proposed in our prior work, is a tunable index with several parameters. For a typical setting of parameters, it can be seen as equivalent to taking a Copy+Log index, and replacing the *snapshot* deltas in it with another set of deltas constructed hierarchically as follows: for every k successive *snapshot* deltas, replace them with a single delta that is the intersection of those deltas and a set of difference deltas from the intersection to the original snapshots, and recursively apply this till you are left with a single delta.

Table 1 estimates the cost of fetching different graph primitives as the number and the cumulative size of deltas that need to be fetched for the different indexes. The first column shows an estimate of index storage space, which varies considerably across the techniques. For proofs, please refer to the extended version [22].

4.3 Temporal Graph Index: Definition

Given the above formalism, a Temporal Graph Index for a graph G over a time period $T = [0, t_c]$ is described by a collection of different deltas as follows:

(a) Eventlist Partitions: A set of eventlist partition deltas, $\{E_{tp}\}$, where E_{tp} captures the changes during the time interval t belonging to partition p.

(b) Derived Snapshot Partitions: Consider r distinct time points, t_i, where $1 \leq i \leq r$, $t_i \in T$, For each t_i, we consider l partition deltas, P^i_j, $1 < j < l$, such that $\cup_j P^i_j = G^{t_i}$. There exists a function that maps any node-id(I) in G^{t_i} to a unique partition-id(P^i_j), $f_i : I \rightarrow P^i_j$. With a collection of P^i_j over T as leaf nodes, we construct a hierarchical tree structure where a parent is the intersection of children deltas. The difference of each parent from its child delta is called as a *derived snapshot partition* and is explicitly stored. Note that P^i_j are not explicitly stored. This is the same as DeltaGraph, with the exception of partitioning.

(c) Version Chain: For all nodes N in the graph G, we maintain a chronologically sorted list of pointers to all the references for that node in the delta sets described above (a and b). For a node I, this is called a *version chain* (denoted VC_I).

In short, the TGI stores *deltas* or *changes* in three different forms, as follows. The first one is the atomic changes in a chronological order through eventlist partitions. This facilitates direct access to the changes that happened to a part or whole of the graph at specified points in time. Secondly, the state of nodes at different points in time is stored indirectly in form of the derived snapshot partition deltas. This facilitates direct access to the state of a neighborhood or the entire graph at a given time. Thirdly, a meta index stores node-wise pointers to the list of chronological changes for each node. This gives us a direct access to the changes occurring to individual nodes. Figure 3(a) shows the arrangement of eventlist, snapshot and derived snapshot partition deltas. Figure 3(b) shows a sample version chain.

TGI utilizes the concept of temporal consistency which was optimally utilized by DeltaGraph. However, it differs from DeltaGraph in two major ways. First, it uses a partitioning for eventlists, snapshots or deltas instead of large monolithic chunks. Additionally, it maintains a list of version chain pointers for each node. The combination of these two novelties along with DeltaGraph's temporal compression generalizes the notion of entity-centric and time-centric indexing approaches in an efficient way. This can be seen by the qualitative comparison shown in Table 1 as well as empirical results in Section 6. Note that the particular design of TGI in the form of eventlist partitions and deltas, and version chain is not equivalent to two separate indexes, one with snapshots and eventlists and the other with chronologically organized events per node. For instance, the latter is fairly inefficient to fetch temporal subgraphs or neighborhoods over time intervals.

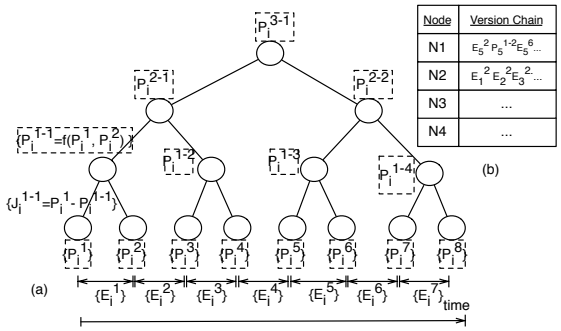


Figure 3: Temporal Graph Index representation: (a) TGI deltas partitions - eventlists, snapshots and derived snapshots. The (dashed) bounded deltas are not stored; (b) Version Chains.

4.4 TGI: Design and Architecture

In the previous subsection, we presented the logical description of TGI. We now describe the strategies for physical storage on a cloud which enables high scalability. In a distributed index, we desire that all graph retrieval calls achieve maximum parallelization through equitable distribution. A distribution strategy based on pure node-based key is good idea for snapshot style access, however, it is bad for a subgraph history style of access. A pure time-based key strategy on the other hand, has complementary qualities and drawbacks. An important related challenge for scalability is dealing with two different skews in a temporal graph dataset – temporal and topological. They refer to the uneven density of graph activity over time and the uneven edge density across regions of the graph, respectively. Another important aspect to note is that for a retrieval task, it is desirable that all the deltas needed for a fetch operation that are present on a particular machine be proximally located to minimize latency of lookups[5]. Based on the above constraints and desired properties, we describe the physical layout of TGI as follows:

1. The entire history of the graph is divided into *time spans*, keeping the number of changes to the graph consistent across different time spans, $f_t : e.time \rightarrow tsid$, where e is the event and $tsid$ is the unique identifier for the time span.

2. A graph at any point is horizontally partitioned into a fixed number of *horizontal partitions* based upon a random function of the node-id, $f_h : nid \rightarrow sid$, where nid is the node-id and sid is unique identifier of for the horizontal partition.

3. The partition deltas (including eventlists) are stored as a key-value pairs, where the delta-key is composed of $\{tsid, sid, did, pid\}$, where did is a delta-id, and pid is the partition-id of the partition.

4. The placement-key is defined as a subset of the composite deltas key described above, as $\{tsid, sid\}$, which defines the chunks in which data is placed across a set of machines on a cluster. A combination of the $tsid$ and sid ensure that a large fetch task, whether snapshot or version oriented, seeks data distributed across the cluster and not just one machine.

[5]In general, this depends on the underlying storage mechanism. The physical placement of deltas is irrelevant for an in-memory store, but significant for an on-disk store due to seek times.

	Index Size	Snapshot		Static Vertex		Vertex versions		1-hop		1-hop Versions	
		$\sum_\Delta \lvert\Delta\rvert$	$\sum_\Delta 1$	$\sum_\Delta \lvert\Delta\rvert$	$\sum_\Delta 1$	$\sum_\Delta \lvert\Delta\rvert$	$\sum_\Delta 1$	$\sum_\Delta \lvert\Delta\rvert$	$\sum_\Delta 1$	$\sum_\Delta \lvert\Delta\rvert$	$\sum_\Delta 1$
Log	$\lvert G\rvert$	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$
Copy	$\lvert G\rvert^2$	$\lvert S\rvert$	1	$\lvert S\rvert$	1	$\lvert S\rvert\lvert G\rvert$	$\lvert G\rvert$	$\lvert S\rvert$	1	$\lvert S\rvert\lvert G\rvert$	$\lvert G\rvert$
Copy+Log	$\frac{\lvert G\rvert^2}{\lvert E\rvert}$	$\lvert S\rvert+\lvert E\rvert$	2	$\lvert S\rvert+\lvert E\rvert$	2	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$	$\lvert S\rvert+\lvert E\rvert$	2	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$
Node Centric	$2\lvert G\rvert$	$2.\lvert G\rvert$	$\lvert N\rvert$	$\lvert C\rvert$	1	$\lvert C\rvert$	1	$\lvert R\rvert.\lvert V\rvert$	$\lvert R\rvert$	$\lvert R\rvert.\lvert V\rvert$	$\lvert R\rvert$
DeltaGraph	$\lvert G\rvert(h+1)$	h.$\lvert S\rvert+\lvert E\rvert$	$2h$	$h.\lvert S\rvert+\lvert E\rvert$	$2h$	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$	$h.(\lvert S\rvert+\lvert E\rvert)$	2h	$\lvert G\rvert$	$\frac{\lvert G\rvert}{\lvert E\rvert}$
TGI	$\lvert G\rvert(2h+3)$	h.$\lvert S\rvert+\lvert E\rvert$	$2h$	$\frac{h.\lvert S\rvert}{p}+\frac{\lvert E\rvert}{p}$	$2h$	$\lvert V\rvert(1+\frac{\lvert S\rvert}{p})$	$\lvert V\rvert+1$	$\frac{h.(\lvert S\rvert+\lvert E\rvert)}{p}$	2h	$\lvert V\rvert(1+\frac{\lvert S\rvert}{p})$	$\lvert V\rvert+1$

Table 1: Comparison of access costs for different retrieval queries and index storage for various temporal indexes. $|G|$ =number of changes in the graph; $|S|$ =size of a snapshot; h = height and $|E|$ = eventlist size; $|V|$ =number of changes to a node; $|R|$=numbers of neighbors of a node; p= number of partitions in TGI. The metrics used are, sum of delta cardinalities ($\sum_\Delta |\Delta|$) and number of deltas ($\sum_\Delta 1$).

5. The partitioned deltas are clustered by the delta key. The given order of delta-key along with the placement-key implies that all partitions of a delta are stored contiguously, which makes it efficient to scan and read all partitions belonging to a delta in a snapshot query. Also, if the order of *did* and *pid* is reversed, it makes fetching a partition across different deltas more efficient.

Implementation and Architecture: TGI uses Cassandra for its delta storage as well as metadata regarding partitioning, time-spans, etc. TGI consists of a *Query Manager* (QM) is responsible for planning, dividing and delegating the query to one or more *Query Processors* (QP). Multiple QPs query the datastore in parallel and process the raw deltas into the required result. Depending on the query specification, the distributed result is either aggregated at a particular QP (the QM) or returned to the client which made the request without aggregation. An *Index Manager* is responsible for the construction and maintenance activities of the index. We omit further details and refer the reader to the extended version [22].

4.5 Dynamic Graph Partitioning

Partitioning of the deltas is an essential aspect of TGI and provides cheaper access to subgraph elements when compared to DeltaGraph or similar indexes. The two traditional approaches to partitioning a static graph are random (node-id hash-based) or locality-based (min-cut, max-flow) partitioning. Random partitioning is simpler and involves minimal bookkeeping. However, since it loses locality, it is unsuitable for neighborhood-level granularity access. Locality-aware partitioning, on the other hand, preserves locality but incurs extra bookkeeping in form of a {node-id:partition-id} map. TGI is designed to work with either configuration as desired, as well as different partition size specifications. TGI also supports replication of edge-cuts for further speed up of 1-hop neighborhoods. It uses a separate *auxiliary delta partition* besides each delta partition to store the replication, thereby preventing extra read cost for snapshot or node centric queries. More details on this can be found in the extended manuscript.

Locality-aware partitioning, however, faces an additional challenge with time-evolving graphs. With the change in size and topology of a graph, a partitioning deemed good (with respect to locality) at an instant may cease to be good at a later time. A probable approach of frequent repartitioning over time would maintain partitioning quality, but leads to excessive amounts of bookkeeping, which in turn leads to degradation of performance while accessing different node or neighborhood versions.

Our approach of dealing with this dilemma is described as follows. For a time-evolving graph, $G(t)$, we update the partitioning once at the beginning of each *time span*. The partitioning valid during a time-span τ, is decided as the collectively best partitioning for the graph during time τ, G^τ. Now, the best-suited partitioning for a graph over a time-interval G^τ is performed by projecting it to a static graph using a function, $\Omega(G^\tau)$, followed by a static-graph partitioning. Ω could be defined in various ways, depending on the best-deemed interpretation of a representative static graph. Any definition, however, must retain all and only the nodes that appeared in G^τ. In TGI, the default choice of Ω is called *Union-Mean* and includes all edges that appeared in G^τ with the edge-weights computed as a function of time-fraction of existence. We refer the reader to the extended manuscript for further details on different choices of Ω, contrast of this technique with other alternatives, and comments on the associated problem of finding the appropriate boundaries of time-spans.

4.6 Fetching Graph Primitives

We briefly describe how the different types of retrieval queries are executed. The details of the algorithms can be found in the extended version of the paper.

Snapshot Retrieval: In snapshot retrieval, the state of a graph at a time point is retrieved. Given a time t_s, the query manager locates the appropriate time span T such that $t_s \in T$, within which, it figures out the path from the root of the TGI to the leaf closest to the given time point. All the snapshot deltas, $\Delta_{s1}, \Delta_{s2}, \ldots, \Delta_{sm}$, (i.e., all the corresponding partitions) along that path from root to the leaf, and the eventlists from the leaf node to the time point, $\Delta_{e1}, \Delta_{e2}, \ldots, \Delta_{en}$ are fetched and merged appropriately as: $\sum_{i=1}^{m} \Delta_{si} + \sum_{i=1}^{n} \Delta_{ei}$ (notice the order). This is performed across different query processors covering the entire set of horizontal partitions. This is conceptually similar to the DeltaGraph snapshot reconstruction with the addition of the aspect of partitions.

Node's history: Retrieving a node's history during time interval, $[t_s, te)$ involves finding the state of the graph at point t_s, and all changes during the time range (t_s, t_e). The first one is done in a similar manner to snapshot retrieval except the fact that we look up only a specific delta partition in a specific horizontal partition, that the node belongs to. The second part happens through fetching the node's version chain to determine its points of changes during the given range. The respective eventlists are fetched and filtered for the given node.

k-hop neighborhood (static): In order to retrieve the k-hop neighborhood of a node, we can proceed in two possible ways. One of them is to fetch the whole graph snapshot and filter the required subgraph. The other is to fetch the given node, and then determine its neighbors, fetch them, and recurse. It is easy to see that the performance of the second method will deteriorate fast with growing k. However for lower values, typically $k \leq 2$, the latter is faster or at least as good, especially if we are using neighborhood replication as discussed in a previous subsection. In case of a neighborhood

fetch, the query manager automatically fetches the auxiliary portions of deltas (if they exist), and if the required nodes are found, further lookup is terminated.

Neighborhood evolution: Neighborhood evolution queries can be posed in two different ways. First, requesting all changes for a described neighborhood, in which case the query manager fetches the initial state of the neighborhood followed by the events indicating the change. Second, requesting the state of the neighborhood at multiple specific time points. This translates to the retrieval of multiple single neighborhoods fetch tasks.

5. ANALYTICS FRAMEWORK

In this section, we describe the *Temporal Graph Analysis Framework (TAF)*, that enables programmers to express complex analytical tasks on time-evolving graphs and execute them in a scalable, parallel, in-memory manner. We present details of the novel computational model, including a set of operators and operands. We we also present the details of implementation on top of Apache Spark, as well as the user API (exposed through Python and Java). Finally, we describe TAF's coordination with TGI, particularly the parallel data transfer protocol, that provides a complete ecosystem for historical graph management and analysis.

5.1 Data and Computational Model

At the heart of this analytics framework is an abstraction with the view of historical graph as a *set of nodes (or subgraphs) evolving over time*. The choice of temporal nodes as a primitive is instrumental in enabling us to express a wide range of fetch and compute operations in an intuitive manner. More significantly, it provides us with the appropriate basis for the parallelizing computation of arbitrary analysis tasks. The *temporal nodes* and *set of temporal nodes* bear a correspondence to *tuples* and *tables* of the relational algebra, as the basic unit of data and the prime operand, respectively. The two central data types are defined below:

DEFINITION 11 (TEMPORAL NODE). *A temporal node (NodeT), N^T, is defined as a sequence of all and only the states of a node N over a time range, $T = [t_s, t_e)$. All the k states of the node must have a valid time duration T_i, such that $\cup_i^k T_i = T$ and $\cap_i^k T_i = \phi$.*

DEFINITION 12 (SET OF TEMPORAL NODES). *A SoN, is defined as a set of r temporal nodes $\{N_1^T, N_2^T \ldots N_r^T\}$ over a time range, $T = [t_s, t_e)$, as depicted in Figure 4.*

The *NodeT* class provides a range of methods to access the state of the node at various time points, including: `getVersions()` which returns the different versions of the node as a list of static nodes (NodeS), `getVersionAt()` which finds a specific version of the node given a timepoint, `getNeighborIDsAt()` which returns IDs of the neighbors at the specified time point, and so on.

A *Temporal Subgraph (SubgraphT)* generalizes NodeT and captures a sequence of the states of a subgraph (i.e., a set of nodes and edges among them) over a period of time. Typically the subgraphs correspond to *k*-hop neighborhoods around a set of nodes in the graph. An analogous `getVersionAt()` function can be used to retrieve the state of the subgraph as of a specific time point as an in-memory `Graph` object (the user program must ensure that any graph object so created can fit in the memory of a single machine). A Set of Temporal Subgraphs (SoTS) is defined analogously to SoN as a set of temporal subgraphs.

5.2 Temporal Graph Analysis Library

The important temporal graph algebra operators supported by our system are described below.

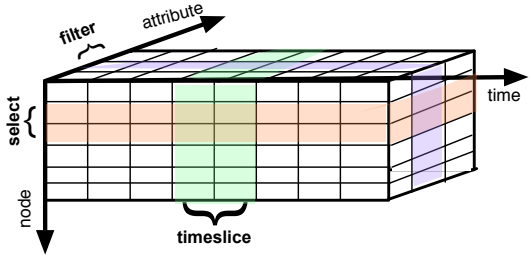


Figure 4: SoN: A set of nodes can be abstracted as a 3 dimensional array with temporal, node and attribute dimensions.

1. ***Selection*** accepts an SoN or an SoTS along with a boolean function on the nodes or the subgraphs, and returns an SoN or SoTS. It performs *entity-centric filtering* on the operand, and does not alter temporal or attribute dimensions of the data.
2. ***Timeslicing*** accepts an SoN or an SoTS along with a timepoint (or time interval) t, finds the state of each of individual nodes or subgraphs in the operand as of t, and returns it as another SoN or SoTS, respectively (SoN/SoTS can represent sets of static nodes or subgraphs as a well). The operator can accept a list of timepoints as input and return a list.
3. ***Graph*** accepts an SoN and returns an in-memory Graph object containing the nodes in the SoN (with only the edges whose both endpoints are in the SoN). An optional parameter, t_p, may be specified to get a GraphS valid at time t_p.
4. ***NodeCompute*** is analogous to a *map* operation; it takes as input an SoN (or an SoTS) and a function, and applies the function to all the individual nodes (subgraphs) and returns the results as a set.
5. ***NodeComputeTemporal***. Unlike `NodeCompute`, this operator takes as input a function that operates on a static node (or subgraph) in addition to an SoN (or an SoTS); for each node (subgraph), it returns a sequence of outputs, one for each different state (version) of that node (or subgraph). Optionally, the user may specify another function (`NodeCompute-Delta`, described next) that operates on the delta between two versions of a node (subgraph). Another optional parameter is a method describing points of time at which computation needs to be performed; in the absence of it, the method will be evaluated at all the points of change.
6. ***NodeComputeDelta*** operator takes as input: (a) a function that operates on a static node (or subgraph) and produces an output quantity, (b) an SoN (or an SoTS) , (c) a function that operates on the following: a static node (or subgraph), some auxiliary information pertaining to that state of the node (or subgraph), the value of the quantity at that state, and an *update* (event) to it. This operator returns a sequence of outputs, one for each state of the node (or subgraph), similar to `NodeComputeTemporal`. However, it computes the required quantity for each version incrementally instead of computing it afresh. An optional parameter is the method describing points of time at which to base the comparison. An optional parameter is a method describing points of time at which computation needs to be performed; in the absence of it, the method will be evaluated at all the points of change.
7. ***Compare*** operator takes as input two SoNs (or two SoTSs) and a scalar function (returning a single value), computes the function value over all the individual components, and returns the differences between the two as a set of *(node-id, difference)* pairs. This operator tries to abstract the common operation of

comparing two different snapshots of a graph at different time points. A simple variation of this operator takes a single SoN (or SoTS) and two timepoints as input, and does the compare on the timeslices of the SoN as of those two timepoints. An optional parameter is the method describing points of time at which to base the comparison.

8. ***Evolution*** operator samples a specified quantity (provided as a function) over time to return evolution of the quantity over a period of time. An optional parameter is the method describing points of time at which to base the evolution.
9. ***TempAggregation*** abstractly represents a collection of temporal aggregation operators such as `Peak`, `Saturate`, `Max`, `Min`, and `Mean` over a scalar timeseries. The aggregation operations are performed over a specified quantity for an SoN or SoTS. For instance, finding "times at which there was a *peak* in the network density" is used to find eventful timepoints of high interconnectivity such as conversations in a cellular network, or high transactional activity in a financial network.

5.3 System Implementation

The library is implemented in Python and Java and is built on top of the Spark API. The choice of Spark provides us with an efficient in-memory cluster compute execution platform, circumventing dealing with the issues of data partitioning, communication, synchronization, and fault tolerance. We provide a GraphX integration for utilizing the capabilities of the Spark based graph processing system for static graphs. Note that while we use Spark for implementation, the concepts presented as a part of the TAF are general and can be implemented over other distributed frameworks such as DryadLINQ[6].

The key abstraction in Spark is that of an RDD, which represents a collection of objects of the same type, stored across a cluster. SoN and SoTS are implemented as RDDs of NodeT and SubgraphT respectively (i.e., as `RDDTG<NodeT>` and `RDDTG<SubgraphT>`, where `RDDTG` extends `RDD` class). Note that the in-memory graph objects may be implemented using any popular graph representation, specially the ones that support useful libraries on top. We now describe in brief the implementation details for NodeT and SubgraphT, followed by details of the incremental computational operator, and the parallel data fetch operation.

Figure 5 shows sample code snippets for three different analytical tasks – (a) finding the node with the *highest clustering coefficient* in a historical snapshot; (b) *comparing different communities* in a network; (c) finding the *evolution of network density* over a sample of ten points.

NodeT and SubgraphT: A set of temporal nodes is represented with an `RDD` of `NodeT` (temporal node). A temporal node contains the information for a node during a specified time interval. The question of the appropriate physical storage of the `NodeT` (or `SubgraphT`) structure is quite similar to storing a temporal graph on disk such as the one using a DeltaGraph or a TGI, however, in-memory instead of disk. Since NodeT is fetched at query time, it is preferable to avoid creating a complicated index, since the cost to create the index at query time is likely to offset any access latency benefits due to the index. Upon observing several analysis tasks, we noticed that the common access pattern is mostly in chronological order, i.e., the query requesting the subsequent versions or changes, in order of time. Hence, we store `NodeT` (and `SubgraphT`) as an initial snapshot of the node (or subgraph), followed by a list of chronologically sorted events. It provides methods such as `GetStartTime()`, `GetEndTime()`, `GetStateAt()`, `GetIterator()`, `Iterator.GetNextVersion()`, `Iterator.GetNextEvent()`, and so on. We omit their details as their functionality is apparent from the nomenclature.

[6]http://research.microsoft.com/en-us/projects/DryadLINQ/

```
tgiH = TGIHandler(tgiconf, "wiki", sparkcontext)
sots = SOTS(k=1, tgiH).Timeslice("t = July 14,2002").fetch()
nm = NodeMetrics()
nodeCC = sots.NodeCompute(nm.LCC, append = True, key="cc")
maxlCC = nodeCC.Max(key="cc")
```

(a) Finding node with highest local clustering coefficient

```
tgiH = TGIHandler(tgiconf, "snet", sparkcontext)
son = SON(tgiH).Timeslice('t >= Jan 1,2003 and t< Jan 1, '
      \',2004').Filter("community")
sonA=son.Select("community =\"A\" ").fetch()
sonB=son.Select("community =\"B\" ").fetch()
compAB = SON.Compare(sonA, sonB, SON.count())
print('Average membership in 2003,')
print(A=%s\tB=%s'%(mean(compAB[0]), mean(compAB[1])))
```

(b) Comparing two communities in a network

```
tgiH = TGIHandler(tgiconf, "wiki", sparkcontext)
son = SON(tgiH).Select("id < 5000").Timeslice("t >= OCt"
      \"24, 2008").fetch()
gm = GraphMetrics()
evol = son.GetGraph().Evolution(gm.density, 10)
print('Graph density over 10 points=%s'%evol)
```

(c) Evolution of network density

Figure 5: Examples of analytics using the TAF Python API.

NodeComputeDelta: `NodeComputeDelta` evaluates a quantity over each NodeT (or SubgraphT) using two supplied methods, $f()$ which computes the quantity on a state of the node or subgraph, and, $f_\Delta()$, which updates the quantity on a state of the node or subgraph for a given set of event updates. Consider a simple example of computing the fraction of all nodes that contain a specific attribute value in a given `SubgraphT`. If this was performed using `NodeComputeTemporal`, the quantity will be computed afresh on each new version of the subgraph, which would cost $\mathcal{O}(N.T)$ operations where N is the size of the operand (number of nodes) and T is the number of versions. However, using incremental computation, each new version after the first snapshot can be processed in constant time, which adds up to $\mathcal{O}(N+T)$. While performing incremental computation, the corresponding $f_\Delta()$ method is expected to be defined so as to evaluate the nature of the event – whether it brings about any change in the output quantity or not, i.e., a scalar change value based upon the actual event and the concerned portions of the state of the graph, and also update the auxiliary structure, if used. Code snippet in Figure 6 illustrates the usage of `NodeComputeTemporal` and `NodeComputeDelta` in a similar example.

Consider a somewhat more intricate example, where one needs to find counts of a small pattern *over time* on an `SoTS`, such as finding the occurrence of a subgraph pattern in the data graph's history. In order to perform such pattern matching over long sequences of subgraph versions, it is essential to maintain certain inverted indexes which can be looked up to answer in constant time whether an event has caused a change in the answer from a previous state or caused a change in the index itself, or both. Such inverted indexes, quite common to subgraph pattern matching, are required to be updated with every event; otherwise, with every new event update, we would need to look up the new state of the subgraph afresh which would simply reduce it to performing non-indexed subgraph pattern matching over new snapshots of a subgraph at each time point, which is a fairly expensive task. In order to utilize a constantly updated set of indices, the auxiliary information, which is a parameter and a return type for $f_\Delta()$, can be utilized. Note that such an incremental computational operator opens up possibilities of utiliz-

```
tgiH = TGIHandler(tgiconf, "dblp", sparkcontext)
sots = SOTS(k=2, tgiH).Timeslice('t >= Nov 1,2009 and t< Nov 30,'\
           '2009').fetch()
labelCount = sots.NodeComputeTemporal(fCountLabel)

labelCount = sots.NodeComputeDelta(fCountLabel, fCountLabelDel)
...
def fCountLabel(g):
   labCount = 0
   for node in g.GetNodes():
      if node.GetPropValue('EntityType') == 'Author':
         labCount += 1
   return labCount

def fCountLabelDel(gPrev, valPrev, event):
   valNew = valPrev
   if event.Type == EType.AttribValAlter:
      if event.AttribKey == 'EntityType':
         if event.PrevVal == 'Author':
            valNew = valPrev - 1
         else if event.NextVal == 'Author':
            valNew = valPrev + 1
   return valNew
```

Figure 6: Incremental computation using different options: `NodeComputeTemporal` and `NodeComputeDelta` to compute counts of nodes with a specific label in subgraphs over time.

ing a large body of algorithmic work in online and streaming graph query evaluation for the purpose of graph analytics.

Specifying interesting time points: In the map-oriented version operators on an `SoN` or an `SoTS`, the time points of evaluation, by default, are all the points of change in the given operand. However, a user may choose to provide a definition of which points to select. This can be as simple as returning a constant set of timepoints, or based on a more complex function of the operand(s). Except the `Compare` operator, which accepts two operands, other operators allow an optional function, which works on a singe temporal operand; the compare accepts a similar function that operates on two such operands. Two such examples can be seen in Figure 7.

```
tgiH = TGIHandler(tgiconf, "wiki", sparkcontext)
son = SON(tgiH).Select("id < 5000").Timeslice("t >= OCt"
        \"24, 2008").fetch()
gm = GraphMetrics()
evol = son.GetGraph().Evolution(gm.density,
     \ selectTimepointsMinimal)
print('Graph density over 3 points=%s'%evol)
...
def selectTimepointsMinimal(son):
    time_arr = []
    st = son.GetStartTime()
    et = son.GetEndTime()
    time_arr.append(st)
    time_arr.append((st + et)/2)
    time_arr.append(et)
    return time_arr
```

Figure 7: Using the optional timepoint specification function for an `Evolution` query with the start, middle and endpoint of SON.

Data Fetch: In a temporal graph analysis task, we first need to instantiate a *TGI connection handler* instance. It contains configurations such as address and port of the TGI *query manager host*, *graph-id*, and a *SparkContext* object. Then, a SON (or SOTS) object is instantiated by passing the reference to the TGI handler, and any query specific parameters (such as k-value for fetching 1-hop neighborhoods with SOTS). The next few instructions specify the semantics of the graph to be fetched from the TGI. This is done through the commands explained in Section 5.1, such as the `Select`, `Filter`, `Timeslice`, etc. However, the actual retrieval from the index doesn't happen until the first statement following the specification instructions. A `fetch()` command can be used explicitly to tell the system to perform the fetch operation. Upon the `fetch()` call, the analytics framework sends the combined instructions to the query planner of the TGI, which translates those instructions into an optimal retrieval plan. This prevents the system from retrieving large amounts of data from the index that is a superset of the required information and prune it later.

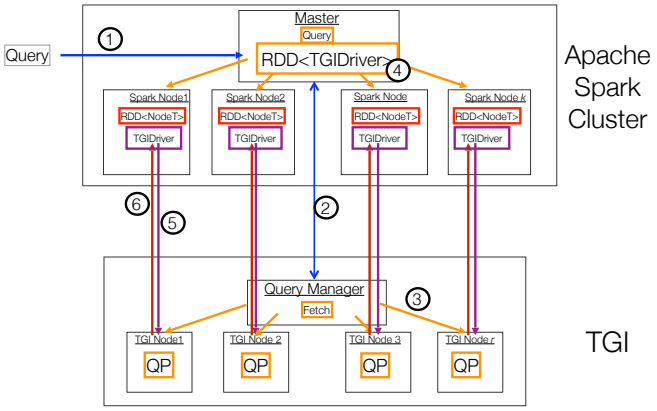


Figure 8: A flow diagram of the parallel fetch operation between the TGI and TAF clusters. The numbers in circles indicate the relative order of events and arrowheads indicate the direction of flow.

The analytics engine runs in parallel on a set of machines, so does the graph index. The parallelism at both places speeds up and scales both the tasks. However, if the retrieval graph at the TGI cluster was aggregated at the Query Manager and sent serially to the master of the analytical framework engine after which it was distributed to the different machines on the cluster, it would create a space and time bottleneck at the Query Manager and the master, respectively, for large graphs. In order to bypass this situation, we have designed a parallel fetch operation, in which there is a direct communication between the nodes of the analytics framework cluster and the nodes of the TGI cluster. This happens through a protocol that can be seen in Figure 8 and summarized below:

1. Analytics query containing fetch instructions is received by the TAF master.
2. A handshake between the TAF master and TGI query manager is established. The latter receives fetch instructions and the former is made aware of the active TGI query processors.
3. Parallel fetch starts at the TGI cluster.
4. The TAF master instantiates a TGIDriver instance at each of its cluster machines wrapped in a RDD.
5. Each node at the TAF performs a handshake with one or more of the TGI nodes.
6. Upon completion of fetch at TGI, the individual TGI nodes transfer the SoN to an RDDs on the corresponding TAF nodes.

More details on the TGI-TAF integration can be found in the longer version of the paper [22].

6. EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the efficiency of TGI and TAF. To recap, TGI is a persistent store for entire histories of large graphs, that enables fast retrieval for a diverse set of graph primitives – snapshots, subgraphs, and nodes at past time points or across intervals of time. We primarily highlight the performance of TGI across the entire spectrum of retrieval primitives. We are not aware of a baseline that may compete with TGI across all or a substantial subset of these retrieval primitives. Specialized alternatives such as DeltaGraph for snapshot retrieval is highly unsuitable for node or neighbor version retrieval; a version centric index may be specialized for node-version retrieval but is highly unsuitable for snapshot

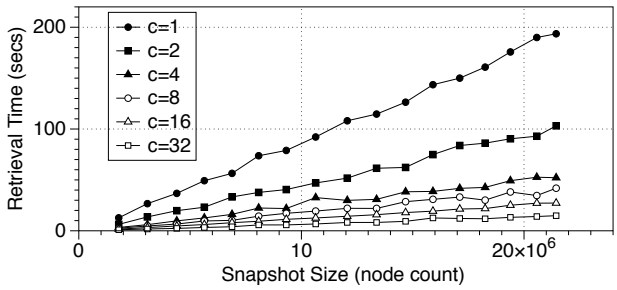


Figure 9: Snapshot retrieval times for varying parallel fetch factor (c), on Dataset 1; $m = 4$; $r = 1, ps = 500$.

or neighborhood-version style retrieval. Also note that TGI generalizes all the known approaches including those two; using appropriate parameter configurations, it can even converge to any specific alternative. Secondly, we demonstrate the scalability of TGI design through experiments on parallel fetching for large and varying data sizes. Finally, we also report experiments demonstrating computational scalability of the TAF for a graph analysis task, as well as the benefits of our incremental computational operator.

Datasets and Notation: We use four datasets: (1) Wikipedia citation network consisting of 266,769,613 edge addition or modification events from Jan 2001 to Sept 2010. At its largest point, the graph consists of 21,443,529 nodes and 122,075,026 edges; (2) We augment Dataset 1 by adding around 333 million synthetic events which randomly add new edges or delete existing edges over a period of time, making a total of 700 million events; (3) Similarly, we add 733 million events, making the total around 1 billion events; (4) Using a Friendster gaming network snapshot, we add synthetic dates at uniform intervals to 500 million events with a total of approximately 37.5 million nodes and 500 million edges.

Following key parameters that are varied in the experiments: data store machine count (m), replication across dataset (r), number of parallel fetching clients (c), eventlist size (l), snapshot or eventlist partition size (ps), and Spark cluster size (m_a).

We conducted all experiments on an Amazon EC2 cluster. Cassandra ran on machines containing 4 cores and 15GB of available memory. We did not use row caching and the actual memory consumption was much lower that the available limit on those machines. Each fetch client ran on a single core with up to 7.5GB available memory. The machines with TAF nodes running Spark workers ran on a single core and 7.5GB of available memory each.

Snapshot retrieval: Figure 9 shows the snapshot retrieval times for Dataset 1 for different values of the parallel fetch factor, c. We observe that the retrieval cost is directly proportional to the size of the output. Further, using multiple clients to retrieve the snapshots in parallel gives near-linear speedup, especially with low parallelism. This demonstrates that TGI can exploit available parallelism well. We expect that with higher values of m (i.e., if the index were distributed across a more machines), linear speedup would be seen for larger values of c (this is corroborated by the next set of experiments). Figure 11c shows snapshot retrieval times for Dataset 4.

Figure 10 shows snapshot retrieval performance for three different sets of values for m and r. We can see that while there is no considerable difference in performance across the different configurations, using two storage machines slightly decreases the query latency over using one machine, in the case of a single query client, $c = 1$. For higher c values, we see that $m = 2$ has a slight edge over $m = 1$. Also, the behavior for the two $m = 1$ and $m = 2; r = 2$ cases are quite similar for same c values. However, we observed that the latter case allows a higher possibility of c value whereas the former peaks out at a lower c value. Further, compression for deltas is negligible for TGI. We omit the detailed points of our investigation, but Figure 11a is representative of the general behavior.

In the special case of $ps \to \infty$, TGI becomes structurally equivalent to a DeltaGraph. While DeltaGraph provides the most efficient way of performing snapshot retrieval, we show that using lower values of ps in TGI only has a marginal impact on the performance of snapshot retrieval (Figure 11b). This occurs due to the TGI design policy of storing all the partitions of a delta contiguously in a cluster and avoiding any additional seek costs. Hence, DeltaGraph is subsumed as a part of TGI and we omit further comparisons in this respect. Also note that the internals of snapshot retrieval through DeltaGraph have been thoroughly explored in our prior work [21].

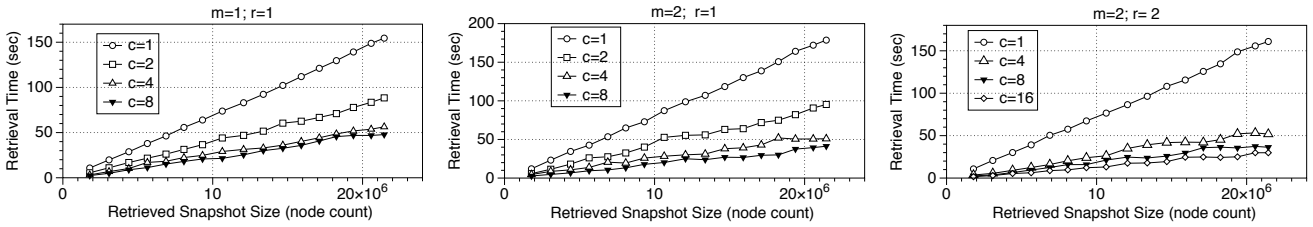
Node History Retrieval: Smaller eventlists or partition sizes provide a lower latency time for retrieving different versions of a node, which can be seen in Figure 12a and Figure 12c, respectively. This is primarily due to the reduction in effort for fetching and deserialization. A higher parallel fetch factor is effective in reducing the latency for version retrieval (Figure 12b). Note that the performances of version and snapshot retrieval for varying partition sizes are opposite. However, smaller eventlist sizes benefit both version and snapshot retrieval. Node version retrieval for Dataset 4 shows a similar behavior, which can be seen in Figure 14.

Neighborhood Retrieval: We compared the performance of retrieving 1-hop neighborhoods, both static and specific versions, using different graph partitioning and replication choices. A topological, flow-based partitioning accesses fewer graph partitions compared to a random partitioning scheme, and a 1-hop neighborhood replication restricts the access to a single partition.This can be seen in Figure 13a for 1-hop neighborhood retrieval latencies. As discussed in Section 4, the 1-hop replication does not affect other queries involving snapshots or individual nodes, as the replicated portion is stored separately from the original partition. In case of a 2-hop neighborhood retrieval, there are similar performance benefits over random partitioning.

Increasing Data Over Time: We observed the fetch performance of TGI with an increasing size of the index. We measured the latencies for retrieving certain snapshots upon varying the time duration of the graph dataset, as shown in Figure 13b. Datasets 2 and 3 contain additional 333 million and 733 million events over dataset 1, respectively. Only a marginal difference in snapshot retrieval performance demonstrates TGI's scalability for large datasets.

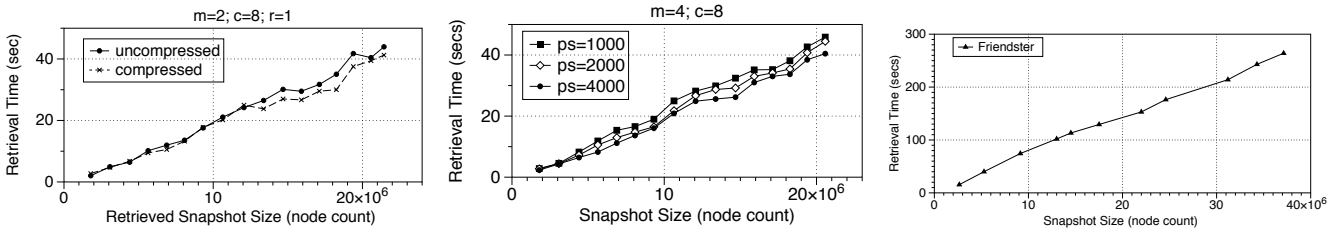
Conducting Scalable Analytics: We examined TAF's performance through an analytical task for determining the highest local clustering coefficient in historical graph snapshot. Figure 13c shows compute times for the given task on different graph sizes, as well as varying size of the Spark cluster. Speedups due to parallel execution can be observed, especially for larger datasets.

Temporal Computation: Earlier in the chapter, we presented two separate ways of computing a quantity over changing versions of a graph (or node). Those include, evaluating the quantity on different versions of the graph separately, and alternatively, performing it in an incremental fashion, utilizing the result for the previous version and updating it with respect to the graph updates. This can be seen for a simple node label counting task in Figure 6. the benefits due to the incremental (`NodeComputeDelta` operator) computation over a version-based computation (`NodeComputeTemporal` operator) can be seen in Figure 15.



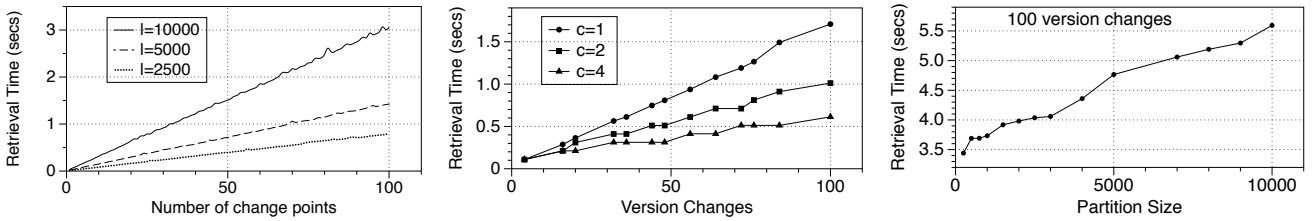
(a) m=1; r=1; ps=500 (b) m=2; r=1; ps=500 (c) m=2; r=2; ps=500

Figure 10: Snapshot retrieval times across different m and r values on Dataset 1.



(a) Compressed vs. uncompressed storage. (b) Effect of partition sizes. (c) Snapshot retrieval; $m = 6$; $c = 1, ps = 500$.

Figure 11: Snapshot retrieval across various parameters. $r = 1$; Dataset 1 for (a) and (b); Dataset 4 for (c).

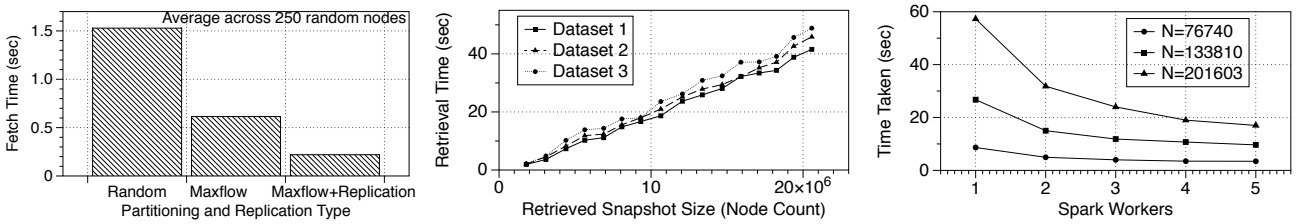


(a) Effect of eventlist size, l. $r = 1$, $c = 1$, $m = 4$, $ps = 500$.

(b) Speedups due to varying parallel fetch factor, c. $m = 4$, $ps = 500$, $r = 1$, $l = 5000$.

(c) Effect of partition sizes.$m = 4$, $r = 1$, $l = 10000$, $c = 1$.

Figure 12: Node version retrieval across various parameters.



(a) Retrieval times for 1-hop neighborhood with different partitioning and replication strategies.

(b) Snapshot retrieval for varying sizes of datasets.

(c) TAF computation times for Local Clustering Coefficient on varying graph sizes (N=node count) using different cluster sizes.

Figure 13: Experiments for partitioning strategies and growing data size ($m = 4$, $r = 2$, $c = 4$, $ps = 500$); TAF analytics computation.

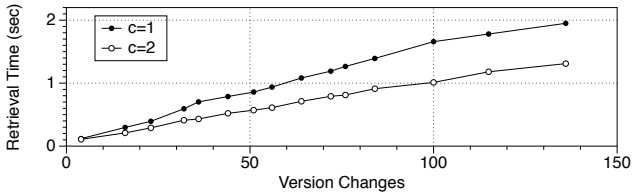


Figure 14: Node versions; Dataset 4; $m = 6$,$r = 1$,$c = 1$,$ps = 500$.

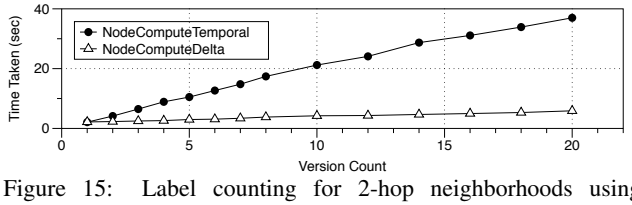


Figure 15: Label counting for 2-hop neighborhoods using (`NodeComputeTemporal`) and (`NodeComputeDelta`), respectively. We report cumulative time taken (excluding fetch time) for varying version counts on Dataset 4 with 2 Spark workers.

7. CONCLUSION

Graph analytics are increasingly considered crucial in obtaining insights about how interconnected entities behave, how information spreads, what are the most influential entities in the data, and many other characteristics. Analyzing the history of a graph's evolution can provide significant additional insights, especially about the future. Most real-world networks however, are large and highly dynamic. This leads to creation of very large histories, making it challenging to store, query, or analyze them. In this paper, we presented a novel Temporal Graph Index that enables compact storage of very large historical graph traces in a distributed fashion, supporting a wide range of retrieval queries to access and analyze only the required portions of the history. Our experiments demonstrate its efficient retrieval performance across a wide range of queries, and can effectively exploit parallelism in a distributed setting. We also presented a distributed analytics framework, built on top of Apache Spark, that allows analysts to quickly write complex temporal analysis tasks and execute them scalably over a cluster.

Acknowledgments: This work was supported by NSF under grant IIS-1319432, an IBM Collaborative Research Award, and an Amazon AWS in Education Research grant.

8. REFERENCES

[1] J.-w. Ahn, C. Plaisant, and B. Shneiderman. A task taxonomy for network evolution analysis. *IEEE Transactions on Visualization and Computer Graphics*, 2014.

[2] L. Arge and J. Vitter. Optimal dynamic interval management in external memory. In *FOCS*, 1996.

[3] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *VLDB*, 2010.

[4] A. Barrat, M. Barthelemy, and A. Vespignani. *Dynamical processes on complex networks*. 2008.

[5] T. Y. Berger-Wolf and J. Saia. A framework for analysis of dynamic social networks. In *SIGKDD*, 2006.

[6] G. Blankenagel and R. Guting. External segment trees. *Algorithmica*, 1994.

[7] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *CloudDB*, 2012.

[8] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EUROSYS*, 2012.

[9] D. Eisenberg, E. M. Marcotte, I. Xenarios, and T. O. Yeates. Protein function in the post-genomic era. *Nature*, 2000.

[10] B. Gedik and R. Bordawekar. Disk-based management of interaction graphs. *TKDE*, 2014.

[11] A. Ghrab, S. Skhiri, S. Jouili, and E. Zimányi. An analytics-aware conceptual model for evolving graphs. In *Data Warehousing and Knowledge Discovery*. 2013.

[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[13] F. Grandi. T-SPARQL: A TSQL2-like temporal query language for RDF. In *ADBIS*, 2010.

[14] D. Greene, D. Doyle, and P. Cunningham. Tracking the evolution of communities in dynamic social networks. In *ASONAM*, 2010.

[15] T. Gross, C. J. D. D'Lima, and B. Blasius. Epidemic dynamics on an adaptive network. *Physical review*, 2006.

[16] R. Gulati and M. Gargiulo. Where do interorganizational networks come from? *American journal of sociology*, 1999.

[17] H. He and A. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.

[18] W. Huo and V. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, 2014.

[19] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *ACM SIGKDD*, 2011.

[20] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In *ACM SIGMOD*, 2013.

[21] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *IEEE ICDE*, 2013.

[22] U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. *CoRR, abs/1509.08960*, 2015.

[23] G. Koloniari and E. Pitoura. Partial view selection for evolving social graphs. In *GRADES workshop*, 2013.

[24] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[25] A. Labouseur, J. Birnbaum, J. Olsen, P., S. Spillane, J. Vijayan, J. Hwang, and W. Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 2014.

[26] K. Lerman and R. Ghosh. Information contagion: An empirical study of the spread of news on digg and twitter social networks. *ICWSM*, 2010.

[27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[28] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays . In *ICDE*, 2015.

[29] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, 2010.

[30] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, et al. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM TOS*, July 2015.

[31] G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: a survey. *IEEE TKDE*, 1995.

[32] R. K. Pan and J. Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 2011.

[33] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *The Semantic Web*. 2006.

[34] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historial evolving graph sequences. In *VLDB*, 2011.

[35] B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 1999.

[36] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD*, 2013.

[37] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *SIGMOD*, 1985.

[38] I. W. Taylor, R. Linding, D. Warde-Farley, Y. Liu, C. Pesquita, D. Faria, S. Bull, T. Pawson, Q. Morris, and J. L. Wrana. Dynamic modularity in protein interaction networks predicts breast cancer outcome. *Nature biotechnology*, 2009.

[39] V. Tsotras and N. Kangelaris. The snapshot index: an I/O-optimal access method for timeslice queries. *Inf. Syst.*, 1995.

[40] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX conference on Hot topics in cloud computing*, 2010.