ABSTRACT

Title of dissertation: HISTORICAL GRAPH DATA MANAGEMENT

Udayan Khurana,
Doctor of Philosophy, 2015

Dissertation directed by: Professor Amol Deshpande
Department of Computer Science

Over the last decade, we have witnessed an increasing interest in *temporal analysis of information networks* such as social networks or citation networks. Finding temporal interaction patterns, visualizing the evolution of graph properties, or even simply comparing them across time, has proven to add significant value in reasoning over networks. However, because of the lack of underlying data management support, much of the work on large-scale graph analytics to date has largely focused on the study of static properties of graph snapshots. Unfortunately, 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. In the absence of appropriate support, an analyst today has to manually navigate the added temporal complexity of large evolving graphs, making the process cumbersome and ineffective.

In this dissertation, I address the key challenges in storing, retrieving, and analyzing large historical graphs. In the first part, I present *DeltaGraph*, a novel, extensible, highly tunable, and distributed hierarchical index structure that enables compact recording of the historical information, and that supports efficient retrieval of historical graph snapshots.

I present analytical models for estimating required storage space and snapshot retrieval times which aid in choosing the right parameters for a specific scenario. I also present optimizations such as partial materialization and columnar storage to speed up snapshot retrieval. In the second part, I present *Temporal Graph Index* that builds upon DeltaGraph to support version-centric retrieval such as a node's *1-hop neighborhood history*, along with snapshot reconstruction. It provides high scalability, employing careful partitioning, distribution, and replication strategies that effectively deal with temporal and topological skew, typical of temporal graph datasets. In the last part of the dissertation, I present *Temporal Graph Analysis Framework* that enables analysts to effectively express a variety of complex historical graph analysis tasks using a set of novel temporal graph operators and to execute them in an efficient and scalable manner on a cloud. My proposed solutions are engineered in the form of a framework called the *Historical Graph Store*, designed to facilitate a wide variety of large-scale historical graph analysis tasks.

Historical Graph Data Management

by

Udayan Khurana

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:

Professor Amol Deshpande	Chair/Advisor
Professor Jimmy Lin	Dean's Representative
Professor Tudor Dumitraş	Committee Member
Dr. Catherine Plaisant	Committee Member
Professor Mihai Pop	Committee Member

Dedication

To my beloved family and friends.

To all the data that my code crunched.

To all the moments that didn't count in the end.

To the hope of a happier planet.

Acknowledgments

My ability to write this dissertation has been in part due to the help, guidance and support from several people. I am grateful to all of them for being a part of my life. Foremost, I want to sincerely thank my advisor Amol Deshpande. His guidance over the course of graduate school greatly helped me in all aspects of my doctoral research. Working with him continuously raised my expectations which was instrumental in the growth of my skills as a researcher. Also, his patience on several occasions helped me work my way through tough times.

I also acknowledge my defense committee members Jimmy Lin, Catherine Plaisant, Mihai Pop and Tudor Dumitras for their constructive suggestions and critiques. My discussions with professors Ben Shneiderman and Nick Roussopoulos over the years were also fruitful in several ways. I am thankful to the department staff, Jenny Story and Fatima Bangura in particular, for their help in all things bureaucratic. My internships at IBM helped me gain a different perspective on research and for that I would like to thank my mentors Srinivasan Parthasarathy and Deepak Turaga. I cherish the time spent with my database lab-mates Ashwin, Theodoros, Jayanta, Waala, Abdul, Hui, Souvik and Amit. Jokes, caricatures and conversations we shared, apart from all the help in form of critiques, motivation and suggestions will be thoroughly missed.

During the long time I spent in Maryland and Washington DC, I met many people who touched my life and made this place feel like home. In particular, I feel fortunate to have met Alice, Hong, Abhishek, Kostas, Sheetal, Sushant, Nakul, Pang, Parul, Kleoniki, Nof and Vikas. I don't want to miss a shout out to those who were miles or oceans away,

but always cared – Sumeet, Mubin, Medha, Joseph, Sandy and Anand.

I want to thank my brother Sukant for always inspiring me with his passion for science. Above all, I want to thank my parents Sarita and Vinod, for being there for me, unconditionally and endlessly. I hope I succeed in imbibing a tiny fraction of the academic quality and personal integrity I see in both of them.

Table of Contents

List of Figures 1

List of Tables 6

1 Introduction 7
- 1.1 Examples of Temporal Networks and Analysis . . . 10
- 1.2 Challenges . . . 16
  - 1.2.1 Storing and fetching temporal graphs . . . 18
  - 1.2.2 Scalable Analytics . . . 22
- 1.3 Contributions . . . 24
  - 1.3.1 DeltaGraph: Reconstructing Past States of a Graph . . . 24
  - 1.3.2 TGI: Generalized Temporal Graph Indexing at Scale . . . 26
  - 1.3.3 Temporal Graph Analytics . . . 27
- 1.4 Graph Data Model . . . 27
- 1.5 Organization of the dissertation . . . 29

2 Background 31
- 2.1 Temporal Graph Analysis . . . 31
  - 2.1.1 Dynamic Social Network Analysis . . . 31
  - 2.1.2 Temporal Network Visualization . . . 32
- 2.2 Temporal Data Management . . . 33
  - 2.2.1 Temporal Queries and Storage . . . 34
  - 2.2.2 Version Management . . . 35
- 2.3 Graph Data Management . . . 36
  - 2.3.1 Graph Storage . . . 36
  - 2.3.2 Graph Query and Processing Frameworks . . . 38
  - 2.3.3 Temporal Graph Data Stores . . . 38

3 Historical Graph Snapshot Retrieval 40
- 3.1 Introduction . . . 40
- 3.2 Background . . . 42
- 3.3 DeltaGraph . . . 45

3.3.1 Anatomy of DeltaGraph . . . 47
3.3.2 Snapshot Queries . . . 49
3.3.3 Accessibility . . . 50
3.4 Snapshot Retrieval . . . 52
3.4.1 Single-point Snapshot Retrieval . . . 52
3.4.2 Multi-point Snapshot Retrieval . . . 53
3.4.3 Composite Time-expression Graph Retrieval . . . 55
3.4.4 Speeding up Snapshot Queries using Prefetching . . . 57
3.4.5 Extensibility . . . 60
3.5 DeltaGraph Analysis . . . 64
3.5.1 Model of Graph Dynamics . . . 64
3.5.2 Differential Functions . . . 65
3.5.3 Space and Time Estimation Models . . . 66
3.5.4 Discussion . . . 69
3.6 Construction and Maintenance . . . 70
3.6.1 Batch Construction . . . 70
3.6.2 Configuring DeltaGraph . . . 72
3.6.3 Updates . . . 74
3.6.3.1 Preliminaries . . . 74
3.6.3.2 Update Procedure . . . 76
3.7 Experiments . . . 79
3.8 Conclusion . . . 90

4 Generalized Historical Graph Storage in the Cloud 92
4.1 Introduction . . . 92
4.2 Temporal Graph Index . . . 95
4.2.1 Preliminaries . . . 96
4.2.2 Prior Techniques . . . 98
4.2.3 TGI Description . . . 100
4.2.4 Scalable TGI design . . . 102
4.2.5 Partitioning and Replication . . . 106
4.2.6 Dynamic Graph Partitioning . . . 109
4.2.7 Fetching Graph Primitives . . . 113
4.3 Evaluation . . . 114
4.4 Conclusion . . . 122

5 Temporal Graph Analytics 125
5.1 Introduction . . . 125
5.1.1 Contributions . . . 126
5.2 Background . . . 127
5.3 Memory Efficient Loading of Multiple Versions . . . 129
5.3.1 GraphPool . . . 130
5.3.2 Clean-up of a graph from memory . . . 132
5.3.3 Visual Analytics on Historical Graph Snapshots . . . 132
5.3.3.1 Features . . . 133

5.3.4 Experiments . . . 138
5.4 Temporal Analytics Framework (TAF) . . . 140
5.4.1 Temporal Graph Analysis Library . . . 140
5.4.2 System Implementation . . . 144
5.4.3 Experiments . . . 153
5.5 Summary . . . 154

6 Conclusion 156
6.1 Insights . . . 158
6.2 Future Directions . . . 160
6.2.1 Unified Temporal Graph Processing Framework . . . 160
6.2.2 Discovering Graphs of Interest . . . 161
6.2.3 A Framework for Finding Temporal Patterns in Evolving Graphs . 163
6.2.4 Shared Computation Across Graph Versions . . . 164
6.2.5 Estimating Changes in Graph Properties . . . 164

Bibliography 165

List of Figures

1.1 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. 12

1.2 The plot was constructed using our system over the DBLP network, and shows the evolution of the nodes ranked in top 25 in 2004. 13

1.3 Performing a snapshot-based evolutionary analysis with and without the support for efficient snapshot retrieval. 17

1.4 A temporal graph can be represented across two different dimensions - time and entity. The chart lists retrieval tasks, graph operations, example queries at different granularities of time and entity size. 20

1.5 Historical Graph Store. 25

3.1 Temporal relational database indexing techniques. 43

3.2 Interval tree is a hierarchical data structure that indexes intervals and can be potentially used to perform snapshot retrieval queries on temporal datasets. 44

3.3 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. 48

3.4 The (Java) code snippet shows an example program that retrieves several graphs, and operates upon them. 51

3.5 Example plans for singlepoint and multipoint retrieval on the DeltaGraph shown in Figure 3.3(a). 54

3.6 Two different options for retrieving difference snapshot queries, $t_1 - t_2$ or $t_2 - t_1$. 58

3.7 Query expression tree for $t_1 \cap t_2 - t_3 \cap t_4$. 58

3.8 The *extensibility* framework of DeltaGraph is meant for users or programmers to execute specific historical queries efficiently by only supplying a logic through *user defined functions* while the framework takes care of the temporal aspects of the implementation. 62

3.9 Different valid DeltaGraphs for input E, configurations, $df = f(); k = 3; l$. All of them have the same number of nodes at each corresponding level, but the structure is different. 75

3.10 Different Packed DeltaGraphs 77

3.11 Summary of Dataset 1 over time. 80

3.12 Summary of Dataset 4 over time. 81

3.13 Comparing DeltaGraph and Copy+Log. *Int* and *Bal* denote the Intersection and Balanced functions, respectively. 82

3.14 Performance of different DeltaGraph configurations vs. Interval Tree for Dataset 2. 83

3.15 Benefits due to multi-core parallelism for snapshot retrieval on a single machine with single disk. 84

3.16 Multipoint query execution vs multiple singlepoint queries. 85

3.17 Retrieval with and without attributes (Dataset 2) 86

3.18 Effect of varying arity (Dataset 1) 87

3.19 Effect of varying eventlist sizes (Dataset 1) 87

3.20 Effect of varying eventlist sizes on snapshot retrieval times (Dataset 4) . . 88

3.21 Snapshot retrieval performance on a DeltaGraph with different eventlist sizes for different time periods (Dataset 2). 88

3.22 Comparison of snapshot retrieval performance for differential functions on Dataset 1. (a) *Intersection* and *Balanced*; (b) Different *Mixed* function configurations. 89

3.23 DeltaGraph snapshot retrieval using single machine, local configurations of Kyotocabinet and Cassandra (Dataset 4). 90

4.1 Temporal Graph Index representation. 101

4.2 The graph history is divided into non-overlapping periods of time. Such division is based on time intervals after which the locality-based graph partitioning is updated. It is also used as a partial key for data chunking and placement. 103

4.3 Graph partitioning using min-cut strategy along with 1-hop replication and the use of auxiliary micro-deltas improves 1-hop neighborhood retrieval performance without affecting the performance of snapshot or node retrieval. 108

4.4 Snapshot retrieval times for varying parallel fetch factor (c), on Dataset 1; $m = 4$; $r = 1, ps = 500$. 116

4.5 Snapshot retrieval times for Dataset 4; $m = 6$; $r = 1, c = 1, ps = 500$. . . 117

4.6 Snapshot retrieval times across different m and r values on Dataset 1. . . 118

4.7 Snapshot retrieval across various parameters. 119

4.8 Node version retrieval across various parameters for Dataset 1. 121

4.9 Node version retrieval for Dataset 4; $m = 6$; $r = 1, c = 1, ps = 500$. . . . 122

4.10 Experiments on partitioning type and replication; growing data size. . . . 123

5.1 (a) Graphs at times $t_{current}$, t_1 and t_2; (b) GraphPool consisting of overlaid graphs; (c) GraphID-Bit Mapping Table 130

5.2 System Architecture: HiNGE, DeltaGraph and GraphPool. 133

5.3 Temporal Exploration using *time-slider* 135

5.4 Temporal Evolution of *node 12* in green over three points in time: Evolution of node's neighborhood and PageRank. 136

5.5 (a) Node Search (b) Subgraph Pattern Search 137
5.6 Cumulative GraphPool memory consumption. 138
5.7 Effect of materialization 139
5.8 SoN: A set of nodes can be abstracted as a 3 dimensional array with temporal, node and attribute dimensions. 141
5.9 Examples of analytics using the TAF Python API. 146
5.10 Incremental computation using different methods. Both examples compute counts of nodes with a specific label in subgraphs over time. 148
5.11 Pattern matching in temporal subgraphs. 149
5.12 Using the optional timepoint specification function with evolution and comparison queries. 150
5.13 A pictorial representation of the parallel fetch operation between the TGI cluster and the analytics framework cluster. The numbers in circles indicate the relative order and the arrowheads indicate the direction of information or data flow. 152
5.14 TAF computation times for Local Clustering Coefficient on varying graph sizes (N=node count) using different cluster sizes. 153
5.15 Label counting in several 2-hop neighborhoods through version (NodeComputeTemporal) and incremental (NodeComputeDelta) computation, respectively. We report cumulative time taken (excluding fetch time) over varying version counts; 2 Spark workers were used for Wikipedia dataset. 154

List of Tables

1.1 Examples of temporal networks. 13

3.1 Options for node attribute retrieval. Similar options exist for edge attribute retrieval. 51

3.2 A list of Differential Functions 66

4.1 Comparison of access costs for different retrieval tasks (snapshot, static vertex, vertex versions, 1-hop neighborhoods and 1-hop neighborhood versions) and index storage on various temporal indexes (Log, Copy, Copy+Log, Node-centric and DeltaGraph). 99

Chapter 1: 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"*, and so on. Such networks, which model the interconnections and interactions between real-world objects, are mathematically represented as *graphs*[1]. Analysis of networks is extensively based on *graph theory*, which provides a wide range of mathematical tools for the study of properties of graphs. Graph theory covers problems such as finding the *shortest path* between two nodes, determining *cliques* and *motifs*, measuring *network flow* between two points, performing *graph coloring*, determining *reachability* and many others [37, 137]; it also describes metrics such as *graph density*, *diameter*, *clustering coefficient*, *pagerank*, to characterize the state of a graph. *Graph mining* applies the principles and methods of graph theory for useful discovery and prediction through *pattern search*, *anomaly detection*, *dense subgraph* detection, *classification*, etc. It finds popular application in several domains such as biology, physics, transportation

[1] Here on, we use the terms *graph* and *network* interchangeably.

engineering, targeted advertising, drug discovery and cyber security. Aggarwal et al. [19] and Chakrabarti et al. [41] are good references for a detailed coverage of techniques and applications of graph mining.

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 network 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 the network since 1997"*, *"how the clusters in the 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 the historical [2] trace of a network in different, usually unpredictable, ways to gain insights into various phenomena. Efforts to perform systematic analysis and visual exploration of such networks over time has prompted work on taxonomies and attempts to define process of analyses for temporal networks [21, 72].

Numerous graph data management systems have been developed over the last decade,

[2] We use the terms *temporal graph* and *historical graph* interchangeably, referring to the chronology of a network's activity from the past, and possibly on-going at the present. We apply the same lack of distinction between *historical data management* versus *temporal data management*, unless otherwise specified. However, we refer to *historical queries* as a subset of the *temporal queries*; the former being static graph queries in the context of a snapshot from the past, whereas the latter being a wider variety including evolution, comparison, historical types, amongst many others.

including specialized graph database systems like Neo4j [10], Titan [15], amongst several others, and large-scale graph processing frameworks such as Pregel [90], Giraph, GraphLab [88], and GraphX [56]. 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. There is recent work on streaming analytics over dynamic graph data [45, 39], but that work typically focuses on analyzing only the recent activity in the network (typically over a sliding window).

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 kinds of relevant historical analyses. 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 simply being able to access past states of the graphs and performing simple static graph analysis, perhaps, 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, we must develop 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. We must also develop an expressive, high-level, easy-to-use programming framework that will allow users to specify complex temporal graph analysis tasks, while ensuring that the specified tasks can be executed efficiently in a data-parallel fashion across a cluster. In this chapter, we proceed by providing a few examples of temporal network analytics from different domains (Section 1.1), followed by the motivation for the work presented in this dissertation and the challenges in providing effective data management solutions for temporal network analytics(Section 1.2), the contributions of this dissertation (Section 1.3), data model for time-evolving graphs (Section 1.4), and finally, an outline of the rest of the dissertation (Section 1.5).

1.1 Examples of Temporal Networks and Analysis

Table 1.1 lists examples of different kinds of temporal networks, such as human interaction networks, biological networks, and others. The temporal component of the network can be based on the change in relationship between two entities, change in the degree of a relationship, or a change in the existence of an entity itself. We present some specific

instances of temporal network analysis in a variety of application domains below:

- **Organizational Sociology:** Sociologists have long studied *inter-organizational alliances* using dynamic network models. For instance, Gulati et al. [60] model the likelihood of a future alliance between industrial organizations or the stability of an existing one, based on dynamic interconnections in a larger network formed over time. Factors such as mutual alliances, and combined centrality over time are important factors in deciding strategic search for alliances.

- **International Finance:** A study of the patterns of flow of Foreign Direct Investment (FDI) in Taiwanese electronic firms [44] shows how certain small firms successfully internationalized themselves by a gradual process of exploiting factors in their networks. The study shows common *temporal trends* with different firms with respect to network activity including strategic *growth of their respective networks*.

- **Epidemiology:** Network modeling has been extensively used in the study and prevention of epidemics as a basis for capturing spatial proximity. In fact, birth of modern epidemiology is credited to John Snow's investigation of the 1854 cholera epidemic of London. Snow [121] elaborated the role of water supply network in the city with empirical evidence of locality and spread of the disease, leading to the discovery that cholera is water-borne. In recent years, there has been a considerable focus on the temporal aspect of networks in epidemiology, in two different ways. The first one is the dynamic nature of networks, i.e., structural changes over time. The other is the epidemic dynamics on network structures, i.e, propagation of infections. Gross et al. [59], present a model for spread of infection based upon a

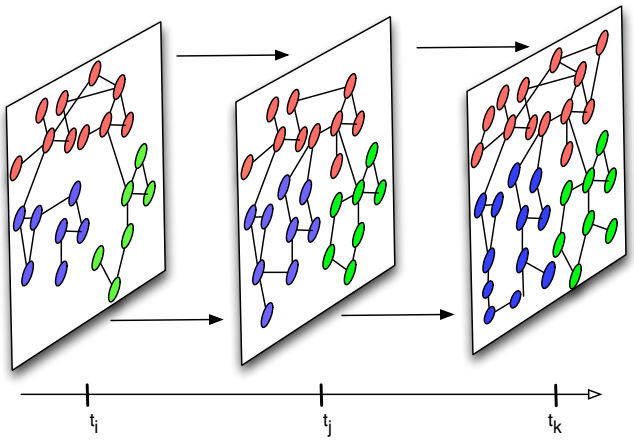


Figure 1.1: Dynamic network analysis (e.g., understanding how "communities" evolve in a social network, how centrality scores change in co-authorship networks, etc.) can lend important insights into social, cultural, and natural phenomena.

changing model of a people network. It exhibits the change of epidemic threshold on different rates of rewiring of links in a simple graph model with constant nodes and number of links. It shows how the adaptive dynamic nature of a physical contact network (along with infection probability) can lead to the state of the epidemic - healthy, oscillatory, bistable or endemic. There have been several other models that explain epidemic growth based on a *random dynamic network* [128, 134].

- **Cellular Networks:** Analysis of social group or community evolution determine crucial characteristics about the community itself. By studying temporal overlap in community structures on scientific collaboration networks and cellular call networks, Palla et al. [101], report that large communities with more dynamic membership last longer than the ones with more rigid membership patterns. However, in the case of smaller communities, the ones with less adaptive membership last longer. Also, their analysis provides a correlation between the time commitment of

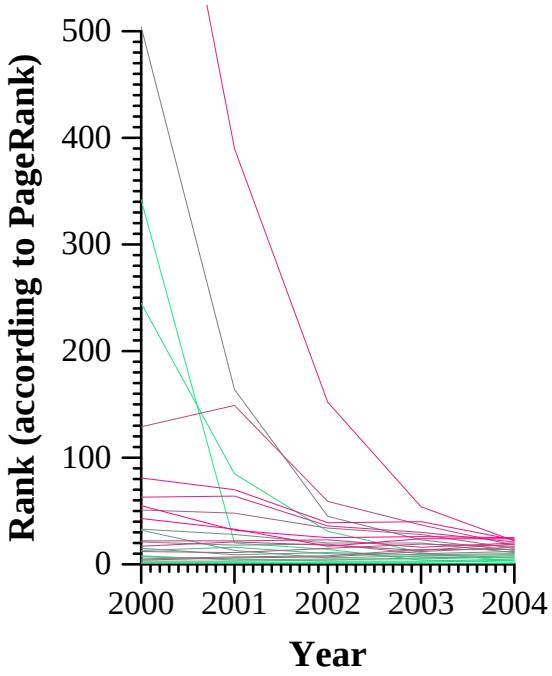


Figure 1.2: The plot was constructed using our system over the DBLP network, and shows the evolution of the nodes ranked in top 25 in 2004.

Category	Source	Entitiy	Relationship	Temporal Aspect
Communication	Email	Person	Email sent	Timestamp of email sent
	Facebook	Person	Friendship	Start (and end) of friendship
	Twitter	Person	Retweet	Retweet timestamp
	Telephone	Person	Conversation	Call duration
Biological	Interactome	Protein molecule	P-P interaction	Order of regulatory or physical relationship
	Metabolism	Enzyme, Substrate	Biochemical reaction	Activity duration
	Cell Signaling Pathway	Protein, Lipid, etc.	Activation, inhibition	Reaction time or order
	Neural	Neuron	Synapse	Spiking / information flow
Transportation	Road map	Intersection	Road segment	Traffic delays
	Postal routes	Post-office, destination	Transfer time	Load variance; frequency
Financial transactional	Bank	Account	Transfer	Time of transfer
	Retail	Merchant, customer	Payment	Purchase or payment time
Computer	WWW	Webpage	Hyperlink	Editing of references
	P2P/ Cluster	Computer	Client-server	Transfer time

Table 1.1: Examples of temporal networks.

members towards a community and the longevity of the community itself.

- **Scientific Collaboration:** Temporal network analysis of the history of bibliographic datasets provides interesting insights into the evolution of collaborative networks or areas of research amongst other things. Erten [50] analyzed a subset of the ACM's digital library to find interesting facts such as: while the typical size of the giant connected component in most domains of science is 80-90% of the collaborative graph, in computing it is just 49%; the average number of co-authors in mathematics has only grown from 1 in 1935 to 1.5 in 1995, exhibiting a slower growth for the same than other fields; in computing, since the mid-nineties, there has been a steady rise of the collaboration networks on topics like data encryption, database management, pattern recognition, but a decline for topics like data structures and symbolic and algebraic manipulation.

- **Biology:** Several biological processes involving interactions between entities are being increasingly modeled as networks. In functional genomics, the classical view of protein molecule functioning focused on the actions of a single protein molecule; with recent advances in the field, the "post-genomic" view of protein functioning focusing on the protein as component of a network of interactive proteins, has gained prominance [49]. Przytycka et al. [105] provide an overview of the recent progress in dynamic protein network modeling and interesting experimental observations. Other areas involving a shift of paradigm towards dynamic networks in biology include cell signaling pathways, gene regulation networks, protein substrate networks and several others. All biological processes are dynamic and involve on-

going change of entity states and their interaction. Analysis of temporal network activity benefits the study and discovery of various biochemical phenomena. For instance, Han et al. [61] study the role of hubs in yeast protein–protein interaction networks for different cellular properties such as robustness of the network. Given the temporal interaction patterns, i.e., whether a hub interacts with other proteins at the same time or at different times, the hubs are classified into two different categories. Each of the two categories are found to contribute towards different functions in the protein-protein interaction network.

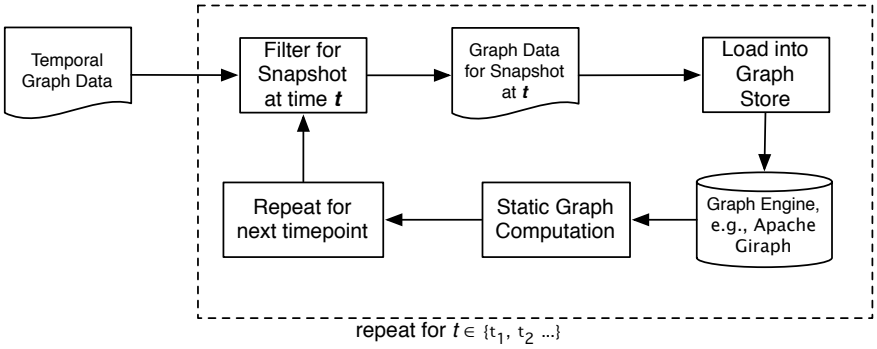
Taylor et al. [127] show that a dynamic model of human protein interaction network (interactome) helps predict breast cancer outcome. They show that the biochemical changes, observed by the *changing structure* of the interactome, convey the relevant information about the disease outcome, which may not be possible in the case of a static protein interaction network.

- **Online Social Networks:** Online social networks such as Facebook, Twitter, Digg, Flickr and others are useful in studying the nature of several aspects of information diffusion as their digital trace is recorded and can be subsequently analyzed. Lerman et al. [84] study the spread of interest in news stories through analyzing the patterns of voting for a story on Digg and retweeting on Twitter. Bakshy et al. [28] study the role of structural components such as few strong links compared to many weak links in the spread of a story on Facebook; they also talk about the differences in the patterns of sharing a story by those who have been exposed to it by their friends and those who haven't been. In both of these works, *temporal clustering*

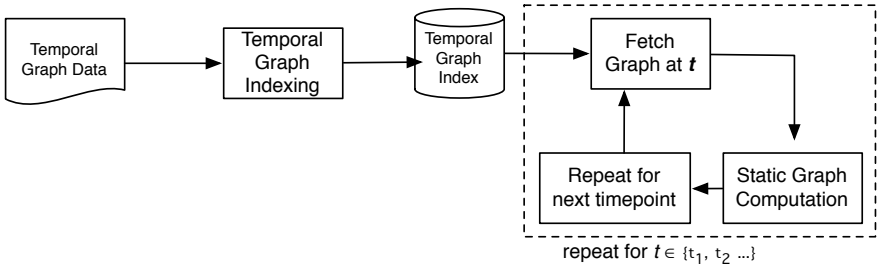
and aggregation was performed along with structural network analysis, in order to study the *flow* or *spread* of information.

1.2 Challenges

In the previous section, we saw that temporal network modeling and analysis finds utility across a wide variety of domains. In order to support a broad range of network analysis tasks, we require a graph data management system at the backend capable of *low-cost storage* and *efficient retrieval* of the historical network information, in addition to providing the correct framework for expressing and executing *analytical 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 to the graph. Hence, it is challenging, often prohibitively so, to perform historical graph analysis at scale due to lack of supporting temporal graph management techniques. For instance, a typical analysis of an evolving graph property through snapshot-based computation over different time points, would require access to several past states of the graph. Figure 1.3(a) illustrates the (simplified) process as of today, where in the absence of a temporal graph store, the analyst is responsible for the work of iteratively filtering the temporal graph data to extract relevant snapshot subsets and ingest them into a "static graph store". This is both burdensome on the analyst as well as time-consuming due to repetitive, often overlapping, ingestion of data. Whereas, while using a "temporal graph store", the analyst would gain access to snapshots promptly on the call of a function, as illustrated in Figure 1.3(b), saving effort and time in the analysis process.



(a) Analysis process using a static graph store



(b) Analysis process using a temporal graph store

Figure 1.3: Performing a snapshot-based evolutionary analysis with and without the support for efficient snapshot retrieval.

The work presented in this dissertation is motivated with the broad goal of enabling an analyst [3] to perform a wide variety of temporal graph data analytics for large evolving networks. Specifically, we found the following aspects of historical graph data management as essential abstractions that needed efficient solutions - (a) compact storage of large graph histories, (b) efficient query time reconstruction of past states of graph or version histories of nodes or neighborhoods, (c) being able to systematically express a variety of temporal graph analysis tasks, hidden from the physical handling of the temporal aspects of data, and (d) a scalable and elastic platform to run such analyses tasks.

Next in this section, we discuss some of the challenges in building an effective

[3] Analysts or a domain experts are typically not equipped with computational skills such as those related to big data management.

historical graph datastore. We first talk about the challenging aspects related to storage and retrieval, followed by issues in performing analytical tasks.

1.2.1 Storing and fetching temporal graphs

The fundamental challenge in storing large evolving graphs is to deal with the complexities of temporal change, as well as the large size of graphs, put together. The primary objective for indexing graph histories is to be able to efficiently fetch (sub-)graph primitives such as snapshots, node or neighborhood versions, effectively while processing analytical queries. Let us first briefly discuss some of the characteristics of static graph storage and temporal indexing, separately, as well as when put together.

- **Compact storage versus fast access:**

  The goal of temporal indexing is twofold. Firstly, it is desired that the storage be compact, i.e., it should incur only a small amount of redundancy, at worst. Secondly, the index should provide fast reconstruction of graph primitives at query time, such as past snapshots, or versions of nodes or neighborhoods. In the known literature on temporal indexing, we see a natural trade-off between compactness and reconstruction efficiency. While a detailed survey of these techniques is presented in Chapter 2 and Chapter 3, two contrasting approaches highlight this duality. First is a "Log" approach, that stores all the changes in a dataset in a chronological order. Without redundancy, this approach occupies minimal space resources. However, reconstruction of, let us say, a past state of the dataset (snapshot), is expensive because there is only a chronological access to any state of the dataset, and no direct

access. This implies that all changes that occurred to the dataset, say attribute values, which are no longer reflected in the snapshot of interest, are information that must be read even if not required to construct the snapshot. On the other hand, a "Copy" approach maintains a distinct copy of the dataset at each change point. This provides direct access to a past snapshot with optimal resource consumption. However, the amount of space consumed is very high because an object that doesn't change across versions is stored multiple times. Conquering this trade-off is the foremost *challenge in temporal indexing*. We state this challenge as, "designing a temporal index structure whose storage consumption is comparable to the Log approach and whose access efficiency is comparable to the Copy approach."

- **Time-centric versus entity-centric indexing:**

  When put together in the same context, temporal data indexing and graph indexing pose further challenges. A temporal graph dataset is based around two dimensions, i.e., *time* and *structural entity*, where the latter represents nodes or edges. If we were to use a storage structure that is primarily indexed by time, such as the Copy approach, it makes it efficient to fetch (snapshots or subsets of) states of the dataset at a desired point in time. However, if the query is based around history of a node, an index based on direct access to states of a dataset is not very helpful. Using it as such implies evaluating all states of the dataset, rendering it highly inefficient, much worse than the Log approach. Alternatively, consider an entity-based temporal index, which separately stores the changes for each entity in a chronological order. While it is suitable for a "node version" type of retrieval, it becomes highly

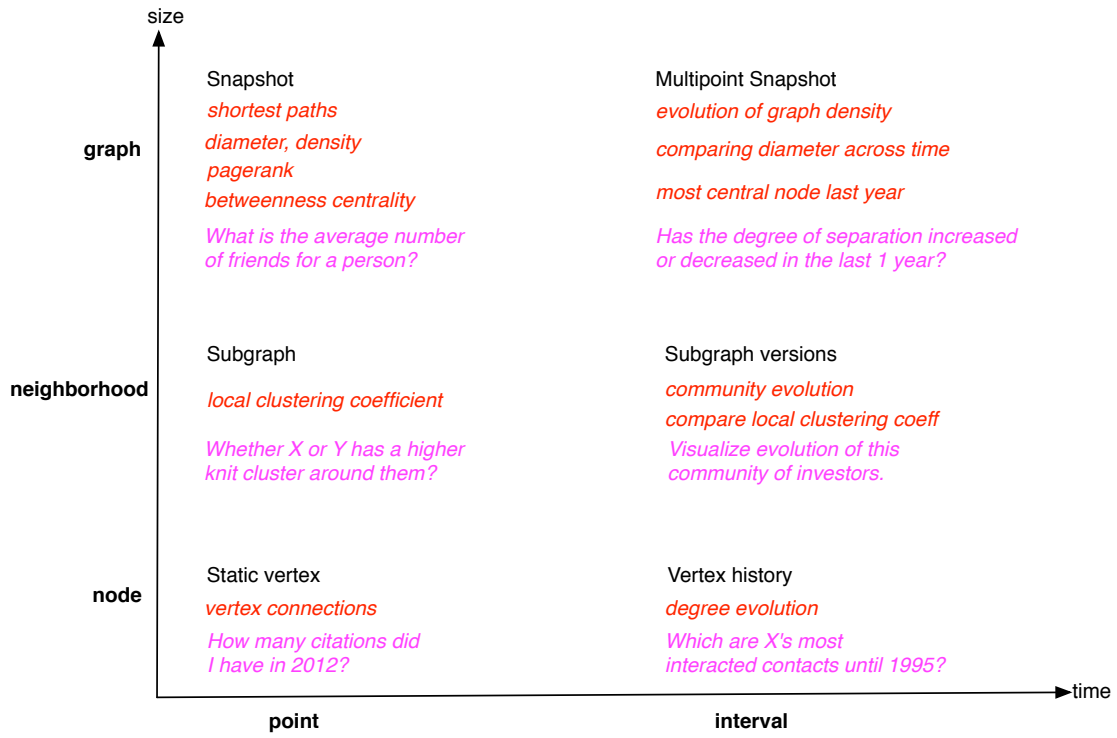


Figure 1.4: A temporal graph can be represented across two different dimensions - time and entity. The chart lists retrieval tasks (black), graph operations (red), example queries (magenta) at different granularities of time and entity size.

inefficient to use it for snapshot retrieval or even neighborhood retrieval. Such a duality of the primacy of time versus the entities in indexing is another fundamental challenge to the problem we address.

- **Coping with changing topology in a dynamic graph:**

  Storing large static graphs in distributed fashion involves careful graph partitioning and replication. However, partitioning graphs does not necessarily provide an ideal solution to distributed graph storage. This is due to the inherent interconnected nature of graphs and a skewed distribution of these connections, i.e., edges, across the set of node pairs. For reasonably dense networks, even optimal partitioning incurs a large number of edge cuts. The poor locality means that for retrieving

typical graph patterns such as *k-hop neighborhoods*, such distributed indexes do not scale very well. In recent years, few systems such as Titan [15] have focused on development of systems for storage of large (static) graphs. Graph partitioning remains an open, hard problem without many effective solutions that work with a wide range of graphs.

The problem of graph partitioning is compounded in the case of changing graphs. As the structure of the graph evolves, so does the best available partitioning over time. While the graph partition may be frequently updated to maintain a good quality of partitioning, it implies heavy amounts of book-keeping, i.e., keeping track of the node–partition mappings, which considerably increases the work in query processing. Because the issue of graph partitioning is essential to any distributed graph management system, the problem of "efficient graph partitioning over evolving graphs" is a fundamental challenge in the context of distributed historical graph data management.

- **Optimal granularity of storage for different queries:**

  Another question pertains to the strategy for physical storage of different components of a graph snapshot. If the snapshot is stored as a single chunk (ignoring the distributed context as of now), it is perhaps the ideal scenario for snapshot retrieval as the entire data to be fetched is stored in a chunk on a disk. However, when this style of storage is used for retrieving individual nodes or 1-hop neighborhoods, it will require fetching the entire snapshot, the bulk of which is not needed. An alternative is the strategy to store the snapshot in smaller chunks. While this will

make fetching smaller portions easier, it also means that fetching a snapshot means looking up for multiple chunks, i.e., multiple seeks, which makes it inefficient. Also, small partitions imply replication of edges (cut during partitioning) in two partitions. Overall increase in space of stored objects also signifies higher retrieval cost. Hence, finding the appropriate physical storage is another challenge in graph indexing.

1.2.2 Scalable Analytics

State-of-the art static network analysis routines cover a wide range of tasks from analyzing graph structure to node properties. Temporal network analytics aims to extend that notion to a temporal dimension. For instance, there is a desire amongst analysts to find the evolution of a certain graph or node metric over time. Additionally, performing temporal aggregates, comparison, or search over time, are other things of prime interest in the realm of temporal graph analytics. In order to define an appropriate framework for it, the following aspects need to be explored:

- **Appropriate abstractions for distributed, scalable in-memory analytics:**

  The major challenge in executing scalable analytics is the parallelization of those tasks for the benefit of fitting the required data into main memory, as well as reducing analysis latencies. Large-scale (static) graph processing frameworks such as Pregel or GraphX do this in the case of static graphs through partitioning the graph by vertices and using a particular message-passing technique underneath to communicate between different nodes. Graph algorithms typically involve accessing

neighbor's information in 1-hop or further distance, which implies a lot of network communication and synchronization. While there is no single model or paradigm of in-memory distributed graph computing accepted as a standard, and it remains an open systems problem, "vertex centric, message-passing" frameworks are considered the current state of the art.

The added *temporal* factor makes this more complicated in the context of temporal graph analytics. The primary question regarding the analytical model is, "what is/are the appropriate abstraction(s) to logically and physically divide a large temporal graph for scalable in-memory analytics?" Whether we partition the data by nodes or by time, or both, needs to be answered appropriately before describing operations on temporal graph data.

- **Systematically expressing temporal graph analytics:**

  A wide variety of analytics on temporal graphs is of interest to analysts and domain specialists. The basis of such analytical tasks lies in graph theory for static graphs. Evolution, comparison or search over time are some of the temporal extension flavors to the static graph quantities. It needs to be determined, "What is the appropriate set of operations that provide sufficient expressive power to analysts/programmers to write various temporal graph analytical tasks?" In other words, "what is the appropriate union of temporal logic and graph algebra?" An additional engineering challenge is to be able to utilize the tools and technology built over the years for static graph analysis, i.e., algorithm libraries such as JUNG, SNAP etc., and cloud computing frameworks such as Spark or MapReduce.

1.3 Contributions

In this dissertation, we present a set of novel techniques to address the different problems in historical graph data management, with the goal of enabling large-scale temporal graph analytics. Broadly, we address the issues of compact storage, reconstruction of past states or histories of nodes, and appropriate framework for large-scale cluster analysis. The contributions are summarized under three different categories in Sections 1.3.1 (DeltaGraph), 1.3.2 (Temporal Graph Index) and 1.3.3 (Temporal Graph Analytics). The different concepts thus developed, have been employed to engineer a system called the *Historical Graph Store (HGS)*. A high level diagrammatic organization of HGS can be seen in Figure 1.5. It consists of two different components. *Temporal Graph Index*, is a persistent and distributed index. It indexes the entire trace of a given historical network (Index Manager) and provides fast access to snapshots, node versions, etc., at query time (Query Manager). *Temporal Graph Analytics Framework* provides a library to specify a variety of analytical tasks, on top of an in-memory cluster-based execution platform. Alternatively, a visual analytics tool, HiNGE, based on GraphPool (an in-memory, compact placeholder for hundreds of graphs simultaneously), may be used. While the cluster-based system is more scalable, HiNGE is memory efficient and can take advantage of traditional graph libraries such as JUNG [99].

1.3.1 DeltaGraph: Reconstructing Past States of a Graph

In order to compactly store the entire historical trace of a network and efficiently reconstruct past states of a network at query time, we present a novel index structure called

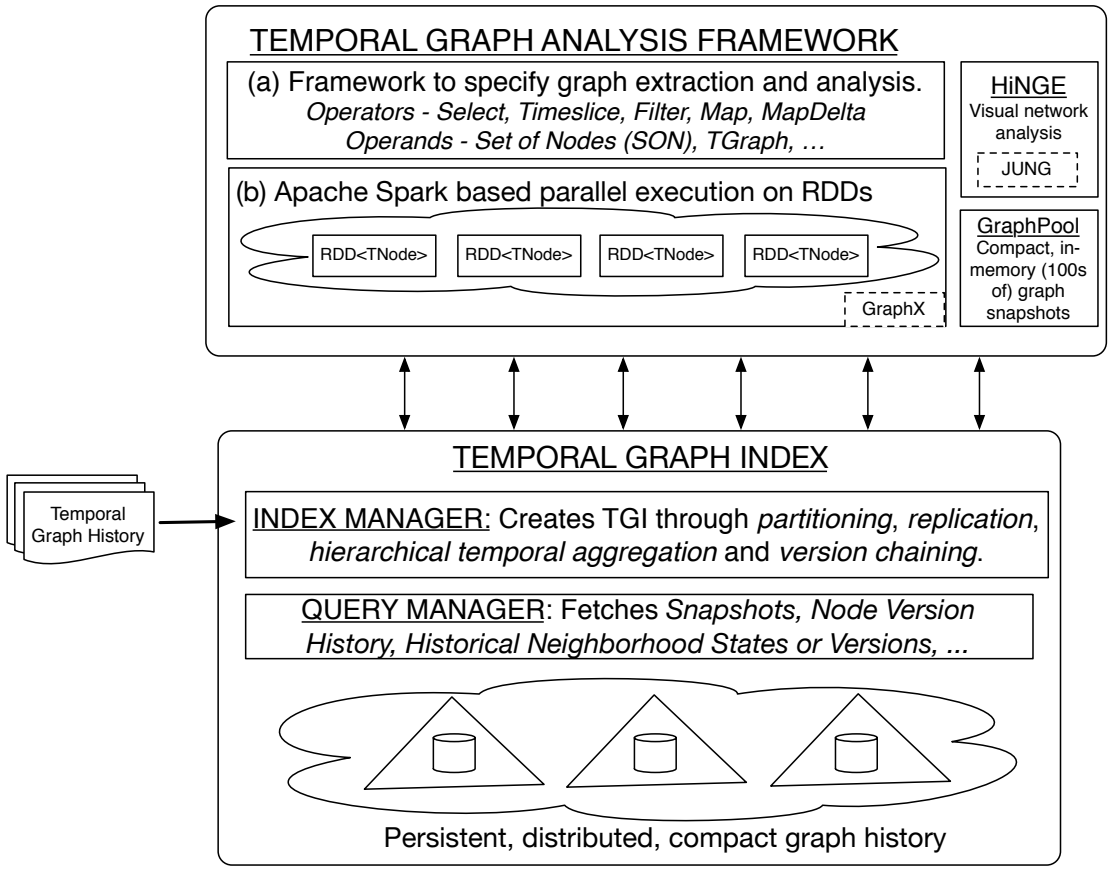


Figure 1.5: Historical Graph Store.

DeltaGraph [73]. It is a hierarchical, distributed, highly tunable and extensible structure geared towards fetching 100s of graph snapshots from disk to memory in milliseconds. The flexibility of DeltaGraph enables us to control the access latencies and storage consumption by tuning simple parameters. While its primary use is in snapshot retrieval, we also developed an *extensibility framework* which helps us to conveniently create different temporal indexes. These temporal indexes can be used to answer specific queries on graph snapshots directly, bypassing the need of explicit snapshot retrieval. Evolutionary analysis requires fetching multiple graph snapshots from different points in time. DeltaGraph's hierarchical structure lends itself into optimizing simultaneous retrieval of

multiple snapshots, exploiting commonality between them. 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.

1.3.2 TGI: Generalized Temporal Graph Indexing at Scale

Although DeltaGraph accomplishes the task of compact storage of historical networks and provides optimal snapshot retrieval, it doesn't support efficient means of answering version-centric queries and its design using snapshot-centric deltas is not entirely suitable for a cloud-oriented elastic store. we developed *Temporal Graph Index (TGI)* [75], an index that builds on DeltaGraph and addresses those two concerns. It compactly stores the entire history of a graph by appropriately *partitioning* and encoding the differences over time (smaller *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 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 provides elastic scalability and a wider variety of graph primitive fetching compared to DeltaGraph. We also show a qualitative comparison between the different indexes, including these two indexes for retrieval of various types of graph primitives.

1.3.3 Temporal Graph Analytics

Since the typical evolutionary analysis queries require fetching 100s of graph snapshots, we need an efficient way to store all of them in memory simultaneously. we developed an in-memory structure called *GraphPool* to store 100s of those graphs in an overlaid manner, exploiting the overlap due to temporal consistency. GraphPool makes it possible to store many snapshots in memory which would otherwise be infeasible. We present a distributed implementation of GraphPool to demonstrate its effectiveness in performing evolutionary and comparative analysis over large evolving graphs. The system exposes a generic programmatic interface as well as an interactive visual extension called, *HiNGE*. Secondly, We present a *Temporal Graph Analysis Framework (TAF)*, which provides an expressive library to specify a wide range of temporal graph analysis tasks and to execute them at scale in a cluster environment. The library is based on a novel set of *temporal graph operators* that enable a user to analyze the history of a graph in a variety of manners. The execution engine itself is based on Apache Spark [142], a large-scale in-memory cluster computing framework.

1.4 Graph Data Model

In this section, we describe a general data model for evolving graphs used in this dissertation. We also consider derivatives of this data model at certain places which are explicitly specified in that context. 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 (assuming 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 deletion followed by a re-insertion results in assignment of a new id). 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.

An *event* is defined 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; examples of a new-edge event (NE) can be seen below, and an update node-attribute event (UNA) as well.

```
(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 G_{k-1} and G_k respectively. If E is the set of all events at time t_k, we have that:

$$G_k = G_{k-1} + E, \quad G_{k-1} = G_k - E$$

where the $+$ and $-$ operators denote application of the events in E in the forward and the backward direction. All events are recorded in the direction of evolving time, i.e., going ahead in time. A list of chronologically organized events is called an *eventlist*.

1.5 Organization of the dissertation

The rest of the dissertation is organized as follows:

- **Chapter 2** discusses references to the relevant background work related to temporal network analysis, graph datastores, and temporal data management. We talk about the cross-disciplinary demand for temporal graph analysis, followed by the limitations and the lack of convergence in existing graph datastores and temporal data management techniques.

- **Chapter 3** describes snapshot retrieval on historical graphs using the DeltaGraph. It contains the design, rationale and analysis of DeltaGraph, a hierarchical persistent index structure, that captures temporal redundancy for compact storage and provides almost direct access to network states of one or more points in the past.

- **Chapter 4** talks about the Temporal Graph Index which provides elastic storage of temporal graphs for a wide-variety of graph primitive retrieval. It builds on the DeltaGraph, but also provides support for version-style retrieval, such as neighborhood evolution. It presents a novel distributed index design that scales temporal or topological skew of a dataset, which is typical of the temporal graphs.

- **Chapter 5** covers two essential aspects of temporal graph analysis through. It talks about a compact way to simultaneously retain hundreds of graphs in memory for analysis using the GraphPool. The latter part of the chapter describes the design and specification of the Temporal Graph Analysis Framework, which provides a library and an execution platform for large-scale temporal graph analytics.

- **Chapter 6** concludes the dissertation through a summary of the observations and contributions, and a description of the future problems related to the topic.

Chapter 2: Background

2.1 Temporal Graph Analysis

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. Social network analysis (SNA) and information visualization (InfoViz) communities in particular have shown great deal of interest around temporal graphs. Following is an account of the recent work in these areas.

2.1.1 Dynamic Social Network Analysis

Many works have focused on designing analytical models that capture how a network evolves, with a primary focus on social networks and the Web (see, e.g., [18, 85, 79]). There is also much work on understanding how communities evolve, identifying key individuals, and locating hidden groups in dynamic networks. Berger-Wolf et al. [32, 125], Tang et al. [123] and Greene et al. [58] address the problem of community evolution in dynamic networks. McCulloh and Carley [92] present techniques for social change detection. Asur et al. [26] present a framework for characterizing the complex behavioral patterns of individuals and communities over time. In a recent work, Ahn et al. [21] present an exhaustive taxonomy of temporal visualization tasks. Biologists are interested in dis-

covering historical events leading to a known state of a Biological network (e.g., [96]). Evolution of shortest paths in dynamic graphs has been studied by Huo et al. [67], Ren et al. [110], and Xuan et al. [62]. Change in page rank with evolving graphs [48, 27], and the study of change in centrality of vertices, path lengths of vertex pairs, etc. [103], also lie under the larger umbrella of temporal graph analysis. Barrat et al. [29], provide a good reference for studying several dynamic processes modeled over graphs. Holme et al. [66] provide a description of several networks from different domains that are temporal or dynamic in nature. 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.

2.1.2 Temporal Network Visualization

Visualization is a principal tool in exploration and analysis of networks. Several network visualization tools such as Gephi [30] and NodeXL [64], are widely used for exploration and analysis of graph datasets. In recent years there has been an emphasis on development of techniques to aid analysis of temporal graphs. These have found application in studying evolution of social networks, Web graph, biological networks, transmission networks etc. Hence, assessing the general data management requirements for such tools is a key motivation in abstracting the correct problems in historical graph data management. Erten et al. [50] present visual analysis for an evolving dataset in computing literature through properties like diameter, connected components, collaborators, co-authors, topic trends over time. Using visualization of different snapshots, Toyoda et al. [131] study events

like appearance of pages, relationship between different pages for a historical dataset of Web archive. More such examples can be found in a survey by Kerracher et al. [72]. Different visualization techniques are useful in the process of gaining insights into the temporal nature of graphs. For instance, NetEvViz [76] extends NodeXL, to study evolving networks. It uses an edge color coding scheme for comparative study of network growth. Ahn et al. [22] focus on different states of graph components in order to visualize temporal dynamics of a social network. Alternate representations to node-link diagrams have been proposed in order to study temporal nature of social networks, such as TimeMatrix [141] that uses a matrix based visualization to better capture temporal aggregations and overlays. A temporal evolution analysis technique based on capturing visual consistency across snapshots can be seen in the work of Xu et al. [140]. A survey by Beck et al. [31] lists several other approaches for visual analysis of dynamic graphs.

2.2 Temporal Data Management

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 a exhaustive survey of that work, but instead refer the reader to several surveys and books on this topic [36, 119, 100, 124, 47, 120, 112]. The most basic concepts that a relational temporal database is based upon are **valid time** and **transaction time**, considered orthogonal to each other. Valid time denotes the time period during which a fact is true with respect to the real world. Transaction time is the time when a fact is stored in the database. A valid-time temporal database permits correction of

a previously incorrectly stored fact [119], unlike transaction-time databases where an inquiry into the past may yield the previously known, perhaps incorrect version of a fact. Under this nomenclature, our data management system is based on valid time – for us the time the information was entered in the database is not critical, but our focus is on the time period when the information is true.

Salzberg and Tsotras [112] 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.Other data structures, such as Interval Trees [25] and Segment trees [34] can also be used for storing temporal information. Temporal aggregation in scientific array databases [122] and time-series indexing [97, 40, 136] are other related topics of interest, but the challenges there are significantly different.

2.2.1 Temporal Queries and Storage

From a querying perspective, both valid-time and transaction-time databases can be treated as simply collections of intervals; however indexing structures that assume transaction times can often be simpler since they don't need to support arbitrary inserts or deletes into the index. Salzberg and Tsotras [112] 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

both, *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, and *valid pure key range* query, where the goal is to fetch an entities states over an interval of time. Snapshot index [132] is an I/O optimal solution to the problem of snapshot retrieval for transaction-time databases. There has also been work on performing *time travel* operations such as obtaining temporal joins, computing temporal aggregates, retrieving version history, etc., [70, 35, 71]. Recently, there was a proposal to add a temporal dimension to the TPC-H benchmark [24]. Many industrial systems, such as Microsoft's ImmortalDB [87], Oracle's Flashback feature [108] and most recently, IBM's DB2 [113] have incorporated one or more of these temporal operations for relational databases.

2.2.2 Version Management

Most scientific datasets are semistructured in nature and can be effectively represented in XML format [38]. Lam and Wong [83] use *complete deltas*, which can be traversed in either direction of time for efficient retrieval. Other systems store the current version as a snapshot and the historical versions as deltas from the current version [91]. For such a system, the deltas only need to be unidirectional. Ghandeharizadeh et al. [54] 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. [38] propose merging all the versions of the database into one single hierarchical data structure for efficient retrieval. In a recent work, Seering et al. [115] presented a disk based versioning system using efficient delta encoding to minimize space consumption

and retrieval time in array-based systems. Bhattacharjee et al. [33] systematically explore the trade-off between storage and recreation costs using compressed delta storage for different versions of a dataset. 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.

2.3 Graph Data Management

In the recent years, there has been much work on graph storage and graph processing systems and numerous commercial and open-source systems have been designed to address various aspects of graph data management. Some examples include Neo4J, AllegroGraph [17], Titan [15], GBase [68], Pregel [90], Giraph [1], GPS [111], Pegasus [69], GraphChi [80], GraphX [56], GraphLab [88], Cassovary [5] and Trinity [117]. 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.

2.3.1 Graph Storage

Massive scale of graph datasets and computational complexity of graph algorithms makes *graph partitioning* and *cluster computing* a common theme across modern graph database. Different graph stores use varying representations for persistent storage of graphs. For instance, GBase stores a distributed sparse adjacency matrix representation of the graph in HDFS files in order to facilitate query processing using Hadoop [2]. Titan, on the other hand uses a different representation utilizing distributed key-value stores such as Cassan-

dra [82], HBase [3] or BerkeleyDB [98]. Other examples of key-value store based graph stores are, CloudGraph [6], VertexDB [16] and Redis Graph [11]. Certain stores such as Neo4J and Titan also feature main memory caching to speed-up graph retrieval from disk.

The query time graph fetch performance, on one hand, depends on the nature of storage, partitioning, chunking, etc. On the other, it also depends on the nature of queries served by the graph store. In general, graph queries follow fairly arbitrary co-access patterns and hence none of the data stores optimize their storage layouts for a broad spectrum of graph queries. For instance, GraphChi ensures serial I/O for a specific pattern of queries for graph mining. Columnar storage is another characteristic increasingly being adopted by various graph stores. It is particularly helpful in decoupling the structural aspects of a graph from the attributes, thereby ensuring lighter fetch payloads as well as greater amount of compression. Columnar storage fits naturally in case of several key-value stores that are designed to handle it effectively.

Relational databases are deemed unsuitable for storing graph data. The inherent nature of traversal by graph queries leads to a high number of expensive joins if the underlying graph data in relational format. However, certain semi-structured data formats such as XML (Extensible Markup Language) and RDF (Resource Description Framework), are able to capture graphs through their data model. RDF stores such as AllegroGraph are able to provide an effective storage base for certain kinds of graph queries with the support of query languages described later. The two prominent reasons for partial success of RDF and XML stores for graph data are, more effective joins in the context of graph traversal and capability to handle fluid schema, respectively.

2.3.2 Graph Query and Processing Frameworks

Most graph querying happens through programmatic access to graphs in languages such as Java, Python or C++. Graph libraries such as Blueprints [4] and SNAP [86] provide a rich set of implementations for graph theoretic algorithms. SPARQL [104, 13] is a language used to search patterns in linked data. It works on an underlying RDF representation of graphs. T-SPARQL [57] is a temporal extension of SPARQL. He et al. [65], provide a language for finding sub-graph patterns using a graph as a query primitive. Gremlin [8] 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.

2.3.3 Temporal Graph Data Stores

There is also prior work on temporal RDF data and temporal XML Data. Motik [95] presents a logic-based approach to representing valid time in RDF and OWL. Several works (e.g., [106, 126]) have considered the problems of subgraph pattern matching or SPARQL query evaluation over temporally annotated RDF data. There has been some work on expressing graph queries in a Datalog-based syntax [116, 129]. There is also much work on version management in XML data stores.

A few recent papers address the issues of storage and retrieval in dynamic graphs. G* [81] stores multiple snapshots compactly by utilizing commonalities. Chronos [63, 94] is an in-memory system for processing dynamic graphs, with objective of shared storage and computation for overlapping snapshots. Ghrab et al. [55] provide a system of network analytics through labeling graph components. Gedik et al. [53], describe a block-oriented and cache-enabled system to exploit spatio-temporal locality for solving temporal neighborhood queries. Koloniari et al. also utilize caching to fetch selective portions of temporal graphs they refer to as partial views [77]. LLAMA [89] uses multi-versioned arrays to represent a mutating graph, but their focus is primarily on in-memory representation.

Chapter 3: Historical Graph Snapshot Retrieval

3.1 Introduction

Snapshot retrieval, or the reconstruction of a past state of the dataset, is perhaps the single most important retrieval primitive for historical network analytics. An evolutionary analysis task such as *finding the evolution of pagerank of the central entities in a network*, involves fetching 100s of snapshots from different time points in the past. Once in memory, we need to run a pagerank routine to compute the required values and plot them. Our ability to to perform this style of historical analysis efficiently depends on our ability to quickly fetch 100s of those snapshots from a persistent datastore to memory. We observe that this crucial task is prevalent across a wide variety of temporal analysis such as community evolution, growth of network characteristics like density, diameter, etc.; it also serves as the basis for answering specific historical queries such as *finding graph clustering coefficient one year ago*. An abstract solution to this problem provides a foundation to the task of storage and retrieval for large historical networks.

In this chapter, we present a novel, user-extensible, highly tunable, and distributed hierarchical index structure called *DeltaGraph*, that compactly stores the entire historical trace of a network, and that supports efficient retrieval of historical graph snapshots for single-site or parallel processing. 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 expose a general programmatic API to process and analyze the snapshots retrieved from DeltaGraph.

Graph Data Model: In the chapter, we follow the data model described in Section 1.4 of the introduction. To recap, we view an evolving graph as a set of snapshots over each time point, where time is assumed to be a discreet quantity. Also, a change between two different snapshots is described by an ordered (chronological) set of events, each of which describes an atomic change in the network.

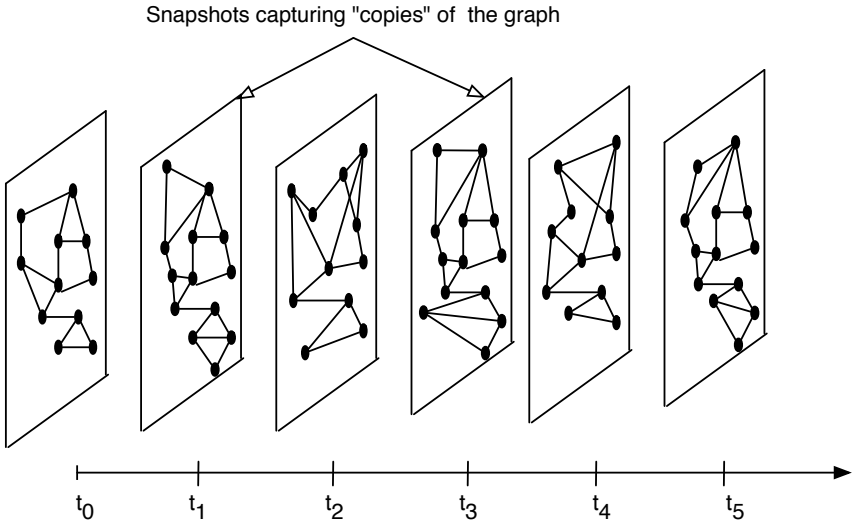
Chapter Outline: In the rest of the chapter, we begin by providing a background of existing alternatives for snapshot retrieval from the temporal relational database, geospatial dataspace and computational geometry literature in Section 3.2. In Section 3.3, we describe the details of DeltaGraph design and an overview of retrieval queries and the accessibility library. Next, we describe the algorithms for performing various snapshot queries in a principled and planned manner in Section 3.4. We also describe query speedup through prefetching and the auxiliary framework for extending DeltaGraph beyond snapshot queries. In Section 3.5, we discuss the impact of different parameters on the behavior of DeltaGraph. We also present the analytical models for estimation of space and query times for given index configurations. In Section 3.6, we discuss the details of the construction and update algorithms for the index. We present experiments supporting the

efficiency of DeltaGraph in Section 3.7 and conclude the chapter in Section 3.8.

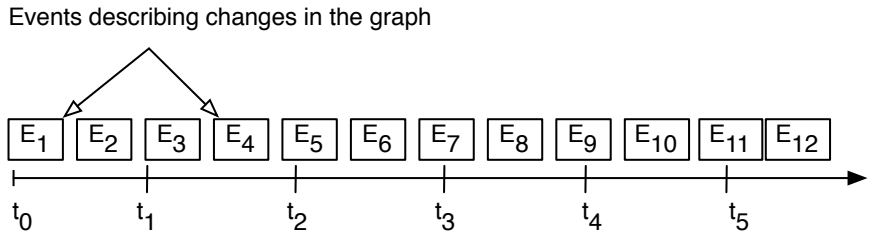
3.2 Background

An optimal solution to answering snapshot retrieval queries is the **external interval tree**, presented by Arge and Vitter [25]. Their proposed index structure uses optimal space on disk, and supports updates in optimal (logarithmic) time. **Segment trees** [34] 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 [132] present **snapshot index**, an I/O optimal solution to the problem for transaction-time databases. Salzberg and Tsotras [112] 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. These are illustrated in Figure 3.1.

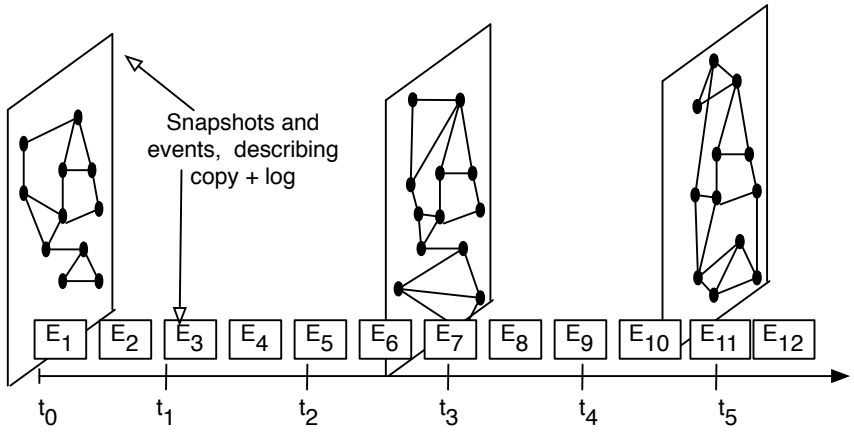
We found these (and other prior) approaches to be insufficient and inflexible for our needs for several reasons. First, they do not efficiently support multipoint queries that we expect to be very commonly used in evolutionary analysis, that need to be op-



(a) Copy



(b) Log



(c) Copy+Log

Figure 3.1: Temporal relational database indexing techniques.

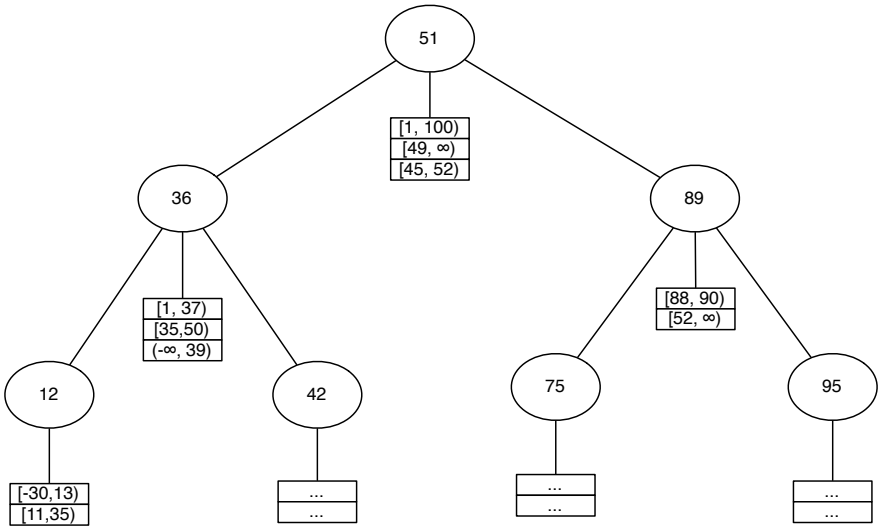


Figure 3.2: *Interval tree* holds intervals and can be potentially used to index a temporal dataset using valid time time durations. Each node contains of: (1) a "center point"; (2) a list of all elements (intervals) whose end points lie across the center points, sorted by start times and end times; (3) pointers to left and right subtrees that correspond to intervals that end before the start of the node's center point and those that start after this node's center point, respectively.

timized by avoiding duplicate reads and repeated processing of the events. Second, to cater to the needs of a variety of different applications, we need the index structure to be highly tunable, and 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. For achieving low latencies, the index structure should support flexible prefetching 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). 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.

3.3 DeltaGraph

Our proposed index data structure, DeltaGraph is a rooted, directed, largely hierarchical graph whose lowest level corresponds to the snapshots 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 graphs 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 very powerful, extensible, general, and tunable index structure that enables trading off the different resources and user requirements as per a specific application's need. By appropriately tuning the DeltaGraph construction parameters, we can trade decreased snapshot retrieval times for increased disk storage requirements. One key parameter of the DeltaGraph enables us to control the distribution of average snapshot retrieval times over history. Portions of the DeltaGraph can be prefetched, allowing us to trade increased memory utilization for reduced query times. Many of these decisions can be made at run-time, enabling us to adaptively respond to changing resource characteristics or user requirements. DeltaGraph is highly extensible, providing a user the opportunity to define additional indexes to be

stored on edge deltas in order to perform specific operations more efficiently. Finally, DeltaGraph utilizes several other optimizations, including a *column-oriented storage* to minimize the amount of 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 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.

We have built a DeltaGraph prototype in Java along with a Python interface. We used Kyoto Cabinet [9], a disk-based key-value store as the back-end engine to store the deltas and eventlists; in the distributed mode, 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 [3] or Cassandra [82].

Finally, we note that our proposed index generalizes beyond graph data and can be used for efficient snapshot retrieval in temporal relational databases as well. In fact, the DeltaGraph treats the network as a collection of objects and does not exploit any properties of the graphical structure of the data.

3.3.1 Anatomy of DeltaGraph

Figure 3.3(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 (equal spacing is not a requirement, but simplifies analysis). 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 $S_{c_1}, \ldots, S_{c_k}$,[1] 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 differential functions in the next section.

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 $\Delta(S_{c_i}, S_p)$ which 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 3.3), 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 3.3). For convenience, we call the children of the super-root as **roots**.

A DeltaGraph can simultaneously have multiple hierarchies that use different dif-

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

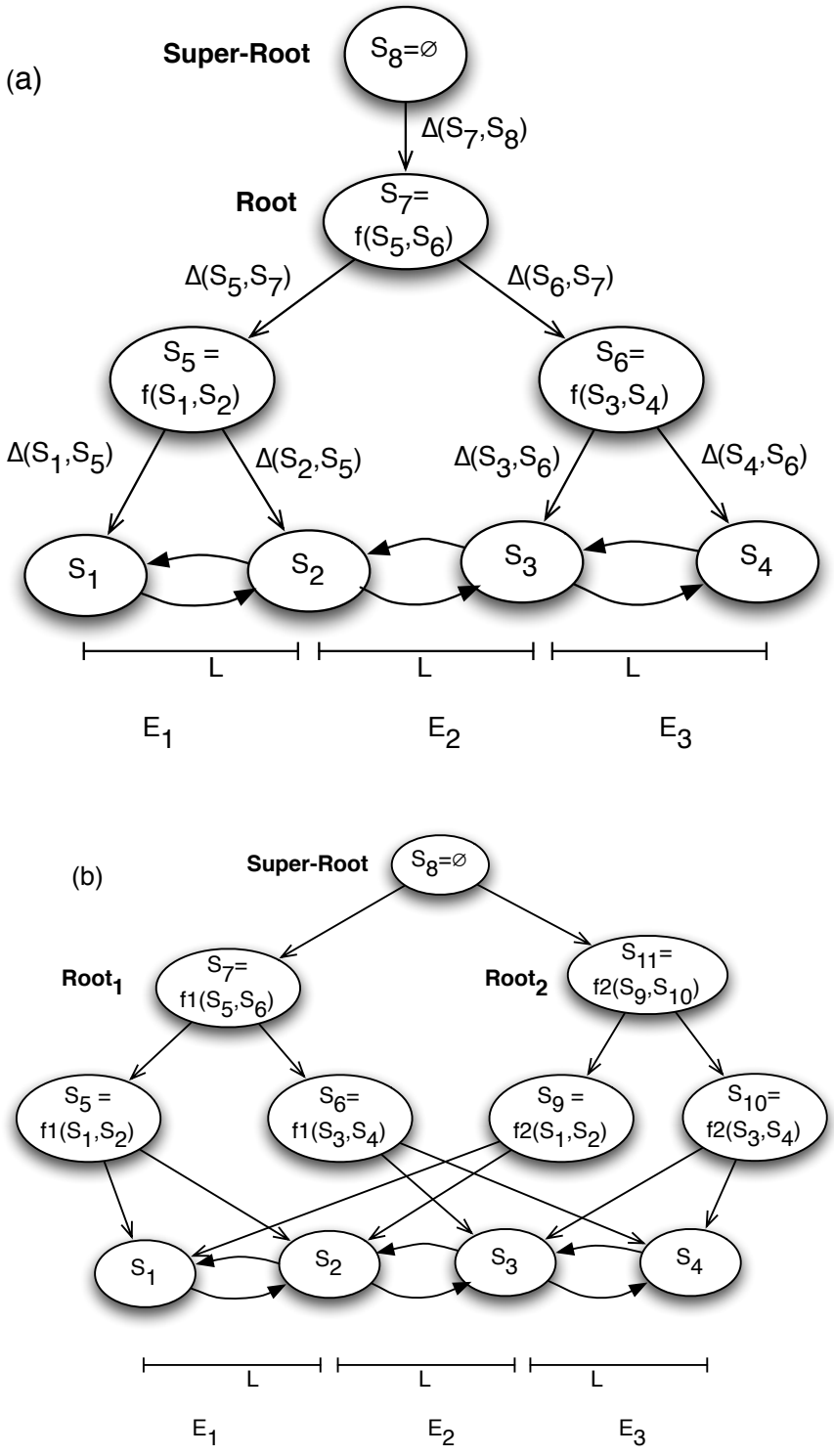


Figure 3.3: DeltaGraphs with 4 leaves, leaf-eventlist size L, arity 2. $\Delta(S_i, S_j)$ denotes delta needed to construct S_i from S_j.

ferential functions (Figure 3.3(b)); this can 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 into three components: (1) Δ_{struct}, (2) $\Delta_{nodeattr}$, and (3) $\Delta_{edgeattr}$. For a leaf-eventlist E, we have an additional component, $E_{transient}$, where the transient events are stored.

Finally, the deltas and the eventlists are stored in a persistent, distributed key-value store, the key being $\langle partition\_id, delta\_id, c \rangle$, where $c \in \{\delta_{struct}, \delta_{nodeattr}, \ldots, E_{transient}\}$ specifies which of the components is being fetched or stored, and $partition\_id$ specifies the partition. Each delta or eventlist is typically partitioned and stored in a distributed manner. Based on the $node\_id$ of the concerned node(s), each event, edge, node and attribute is designated a partition, using a hash function h_p, such that, $partition\_id = h_p(node\_id)$. In a set up with k distributed storage units, all the deltas and eventlists are likely to have k partitions each.

3.3.2 Snapshot Queries

We differentiate between a *singlepoint 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.

3.3.3 Accessibility

The system can be used through a programmatic API, an interactive visual interface for analytics or a declarative high level query language. The programmatic API is the best way to interact with the system in a tightly coupled manner. 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): In the basic singlepoint graph retrieval call, the first parameter indicates the time; the second parameter indicates the attribute information to be fetched as a string formed by concatenating sub-options listed in Table 3.1. For example, to specify that all node attributes except *salary*, and the edge attribute *name* should be fetched, we would use: attr_options = "+node:all-node:salary+edge:name".

GetHistGraphs(List<Time> t_list, String attr_options), where t_list specifies a list of time points.

GetHistGraph(TimeExpression tex, String attr_options): This is used to retrieve a hypothetical graph using a multipoint expression over time points. For example, the expression $(t_1 - t_2)$ specifies the components of the graph that were valid at time t_1 but not at time t_2. The TimeExpression data structure consists of a list of k time points, $\{t_1, t_2, \ldots, t_k\}$, and binary set operators, union ($\cup$), intersection ($\cap$) and subtract ($-$) over them.

```
/* Loading the index */
GraphManager gm = new GraphManager(. . . );
gm.loadDeltaGraphIndex(. . . );
...
/* Retrieve the historical graph structure along with node names
and Jan 2, 1985 */

HistGraph h1 = gm.GetHistGraph("1/2/1985", "+node:name");
...
/* Traversing the graph*/
List<HistNode> nodes = h1.getNodes();
List<HistNode> neighborList = nodes.get(0).getNeighbors();
HistEdge ed = h1.getEdgeObj(nodes.get(0), neighborList.get(0));
...
/* Retrieve the historical graph structure alone on Jan 2, 1986
and Jan 2, 1987 */

listOfDates.add("1/2/1986");
listOfDates.add("1/2/1987");
List<HistGraph> h1 = gm.getHistGraphs(listOfDates, "");
...
```

Figure 3.4: The (Java) code snippet shows an example program that retrieves several graphs, and operates upon them.

GetHistGraphInterval(Time t_s, Time t_e, String attr_options):

This is used to retrieve a graph over all the elements that were added during the time interval $[t_s, t_e)$. It also fetches the transient events, not fetched (by definition) by the above calls.

Option	Explanation
-node:all (**default**)	None of the node attributes
+node:all	All node attributes
+node:attr1	Node attribute named "attr1"; overrides "-node:all" for that attribute
-node:attr1	Node attribute named "attr1"; overrides "+node:all" for that attribute

Table 3.1: Options for node attribute retrieval. Similar options exist for edge attribute retrieval.

3.4 Snapshot Retrieval

3.4.1 Single-point Snapshot Retrieval

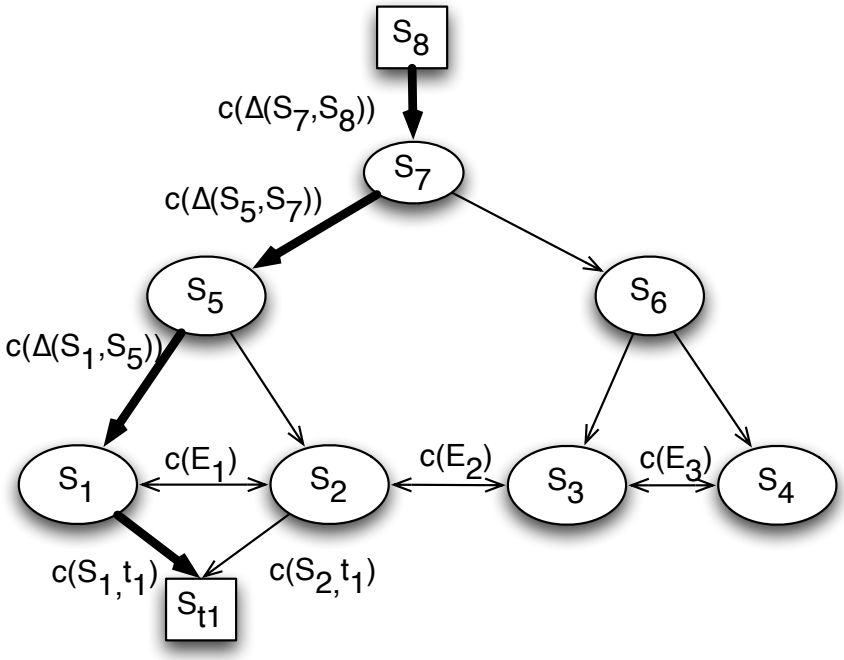
Given a singlepoint snapshot query at time t_1, there are many ways to answer it from the DeltaGraph. Let E denote the leaf-eventlist such that $t_1 \in [E^{start}, E^{end})$ (found through a *binary search* at the leaf level). Any (directed) path from the super-root to the two leaves adjacent to E represents 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 four 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 3.5(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. Although this algorithm requires us to traverse the entire DeltaGraph skeleton for every query, it is needed to handle the continuously changing DeltaGraph skeleton, especially in response to **memory materialization** (discussed later). Second, the weights associated with the edges are different for different queries and the weights are also highly skewed, so the shortest

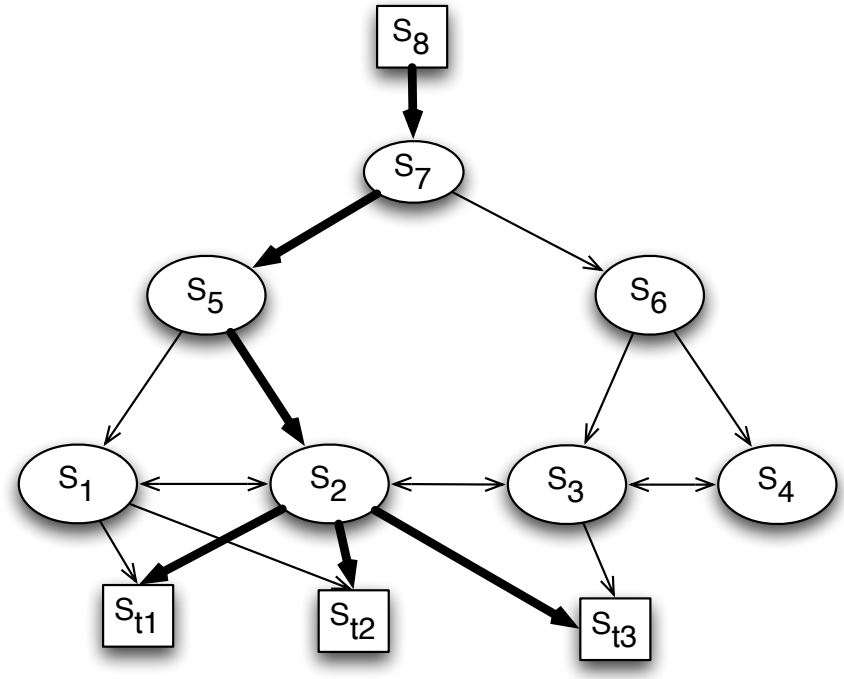
paths can be quite different for the same timepoint for different attr_options. Further, the sizes of the DeltaGraph skeletons (i.e., the number of nodes and edges) are usually small, even for very large historical traces, and the running time of the shortest path algorithm is dwarfed by the cost of retrieving the deltas from the persistent storage. In ongoing work, we are working on developing an algorithm based on incrementally maintaining single source shortest paths to handle very large DeltaGraph skeletons.

3.4.2 Multi-point Snapshot Retrieval

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 3.3(a). We first identify the leaf-level eventlists that contain the three time points, and add virtual nodes S_{t1}, S_{t2}, S_{t3}, shown in the figure using shaded nodes. 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 does 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 algorithm [78] for undirected Steiner trees for that purpose. 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



(a) Singlepoint query t_1



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

Figure 3.5: Example plans for singlepoint and multipoint retrieval on the DeltaGraph shown in Figure 3.3(a).

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.

3.4.3 Composite Time-expression Graph Retrieval

A composite time-expression can be used to retrieve a graph more complex than snapshot(s). For instance, *"changes to a network that happened between January and July of 2013"*, which can be symbolically written as $G(t_1) - G(t_2)$, or simply, $G(t_1 - t_2)$. One way to perform composite time-expression retrieval queries is to execute a multi-point snapshot query to fetch the required snapshots into memory and then operating upon them to find the components that satisfy the TimeExpression.

However, we present more efficient ways to address such composite graph queries. We provide a set based expression of complex time expression using the operators: *Union*, *Intersection* and *Difference*.

For two valid time points in a graphs history, t_1 and t_2 , where $t_1 \neq t_2$:

- $t_1 - t_2$: The difference of two snapshots, one at t_2 from another at t_1, contains the (portions of) nodes in t_1 which are not present in t_2. For example, if a node existed in both snapshots, but had a few edgelist elements or attributes key-value pairs that were only in t_1 but not in t_2, that node with those exclusive components present in t_1 appear in the result. However, if the state of a node is present as it is

in both snapshots, that node doesn't appear in the difference. In order to perform the difference of two snapshots, we traverse from one time point to the other, t_1 to t_2 and read the events on the way. If $t_1 < t_2$, then we take up all the *deletion events* and construct the graph using those events, ignoring the deletion instruction. We ignore the addition events in this case. In the case of $t_1 > t_2$, we consider the addition events instead of the the deletion events. The other way of performing this operation would be to fetch the two snapshots at t_1 and t_2, respectively and perform the set difference operation. The two point snapshot retrieval can be made using the steiner tree formulation. Both the alternatives are illustrated in Figure 3.6. Depending on different cases, one may be more efficient than the other. In terms of query planning, we evaluate the cost of both plans and then choose the cheaper one.

- $t_1 \cup t_2$: The straightforward way of performing a union of two snapshots at t_1 and t_2 is to fetch both the snapshots and perform an element by element union. The other way is to retrieve one snapshot, say at t_1 and traverse the events from there to t_2, and adding all the *addition events* to the retrieved snapshot from t_1. The fetch cost involved in the second case can be equal or more than the fetch cost involved in the first. The case when both are the same is the one when when the steiner tree involves all the events from t_1 to t_2 as well. However, the advantage of using the second one is lesser work involved in performing the union in most cases. The amount of work done in performing the union in the first case is $(|G(t_1)| + |G(t_2)|)$ can be up to two times the size of the output $(|G(t_1 \cup t_2)|)$. The work done in the

second case is the number of addition events from t_1 to t_2, which is at most the total events in between the two, $E_{t1 \rightarrow t2}$. Depending on the cost estimates of the bounds, the planner makes a decision on how to execute the query. Typically, for snapshots close to each other, the second way is preferable, due to small eventlists. For snapshots much far apart, it depends on the composition of the eventlist.

- $t_1 \cap t_2$: Similar to the procedures described for performing the union above, intersection of two snapshots, corresponding to t_1 and t_2 ($t_1 < t_2$) can also be performed in two different ways. First, fetching the snapshots corresponding to t_1 and t_2, and then performing the intersection operation. Second, fetching the snapshot at t_1, followed by traversing from t_1 to t_2 and applying all the negative events (note the difference from the union here). If $t_1 > t_2$, we apply the addition events as deletion events instead. The query planning makes the decision on which plan to choose based upon the estimated costs.

For more complex expressions, that may use a complex combination of these operators, we sketch a query expression tree and perform the first level (in the tree) operations using the methods specified above; the higher level operators using a direct computation on the resulting operands from the first level operations, such as the one illustrated in Figure 3.7.

3.4.4 Speeding up Snapshot Queries using Prefetching

For improving query latencies, some nodes in the DeltaGraph are typically prefetched and *materialized* in memory. The process of "materialization" also involves an in-memory

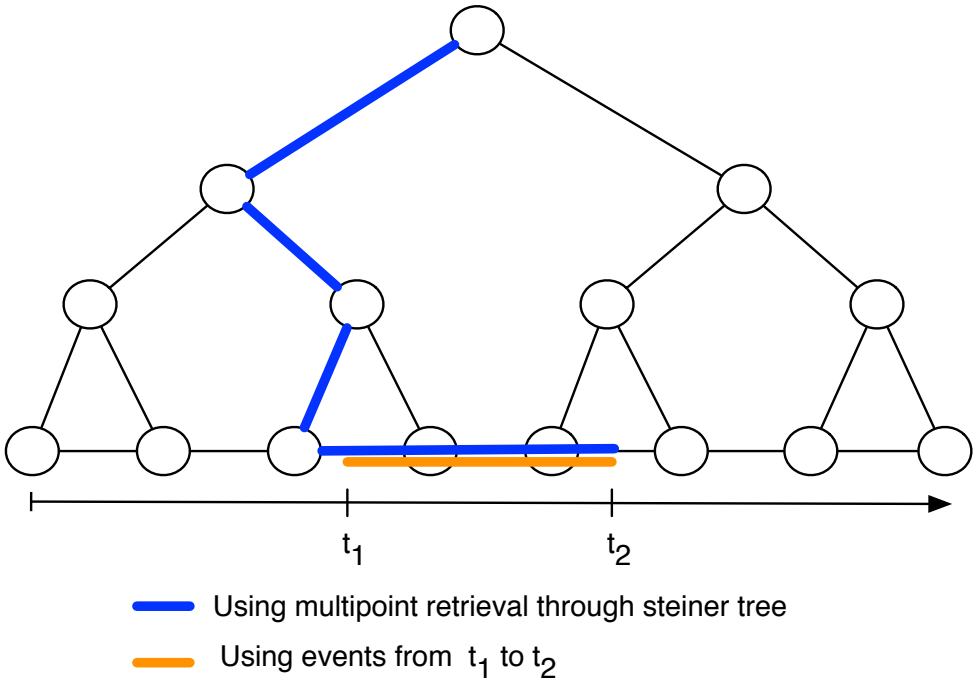


Figure 3.6: Two different options for retrieving difference snapshot queries, $t_1 - t_2$ or $t_2 - t_1$.

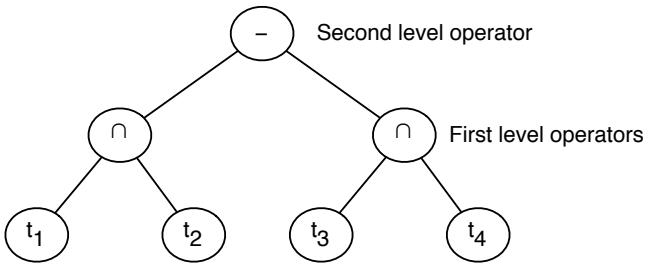


Figure 3.7: Query expression tree for $t_1 \cap t_2 - t_3 \cap t_4$.

structure for storing multiple graphs in an overlaid manner, called GraphPool (explained in Chapter 5). However, at this point, it is not essential to know the specifics of GraphPool to understand the benefits of materialization. For now, we refer to the prefetching of a (leaf or internal) snapshot in memory and using that while processing a snapshot retrieval query that benefits from materialization. In particular, the highest levels of the DeltaGraph should be materialized, because it benefits retrieval for snapshots across the board. And further, the "rightmost" leaf (that corresponds to the current 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 need 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.

3.4.5 Extensibility

To efficiently support specific types of queries or tasks, it is beneficial to maintain and index auxiliary information in the DeltaGraph and use it during query execution. We extend the DeltaGraph functionality through user-defined modules and functions for this purpose. 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. The user may also supply a different differential function to be used for this auxiliary information. The basic retrieval functionality (i.e., retrieve snapshots as of a specified time points) is thus naturally extended to such auxiliary information, which can also be loaded into memory and operated upon. 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.

The extensibility framework involves the *AuxiliarySnapshot* and *AuxiliaryEvent* structures which are similar to the graph snapshot and event structures, respectively. An AuxiliarySnapshot consists of a hashtable of string key-value pairs where as an AuxiliaryEvent consists of the event timestamp, a flag indicating the addition, deletion or the change of a key-value pair, and finally, a key-value pair itself. Given the very general nature of the auxiliary structures, it is possible for the consumer (programs using this API) to define a wide variety of graph indexing semantics whose historical indexing will be done automatically by the HistoryManager. For a historical graph, any number of auxiliary indexes may be used, each one by extending the *AuxIndex* abstract class, defining the following, a) method *CreateAuxEvent* that generates an AuxiliaryEvent correspond-

ing to a plain Event, based upon the current Graph and the latest Auxiliary Snapshot, b) method *CreateAuxSnapshot*, to create a leaf-level AuxiliarySnapshot based upon the previous AuxilarySnapshot and the AuxiliaryEventList in between the two, and c) a method *AuxDF*, a differential function that computes the parent AuxiliarySnapshot, given a list of k AuxiliarySnapshots and corresponding $k-1$ AuxEventlists.

Any number of queries may be defined on an auxiliary index by extending either of the three *Auxiliary Query* abstract classes, namely, *AuxHistQueryPoint*, *AuxHistQueryInterval* or *AuxHistQuery*, depending on the temporal nature of the query - point, range or across entire time span, respectively. An implementation of any of these classes involves attaching a pointer to the AuxiliaryIndex which the queries are supposed to operate upon (in addition to the plain DeltaGraph index itself), and the particular query method, *AuxQuery*. While doing so, the the programmer typically makes use of the methods like *GetAuxSnapshot* provided by the Extensibility API.

We illustrate this extensibility through an example of a *subgraph pattern matching* index. Techniques for subgraph pattern matching are very well studied in literature (see, e.g., [65]). Say we want to support finding all instances of a node-labeled query graph (called *pattern graph*) in a node-labeled data graph. One simple way to efficiently support such queries is to index all paths of say length 4 in the data graph [118]. This pattern index here takes the form of a *key-value* data structure, where a *key* is a quartet of labels, and the value is the set of all paths in the data graph over 4 nodes that match it. To find all matching instances of a pattern, we decompose the pattern into paths of length 4 (there must be at least one such path in the pattern), use the index to find the sets of paths that match those decomposed paths, and do an appropriate join to construct the entire match.

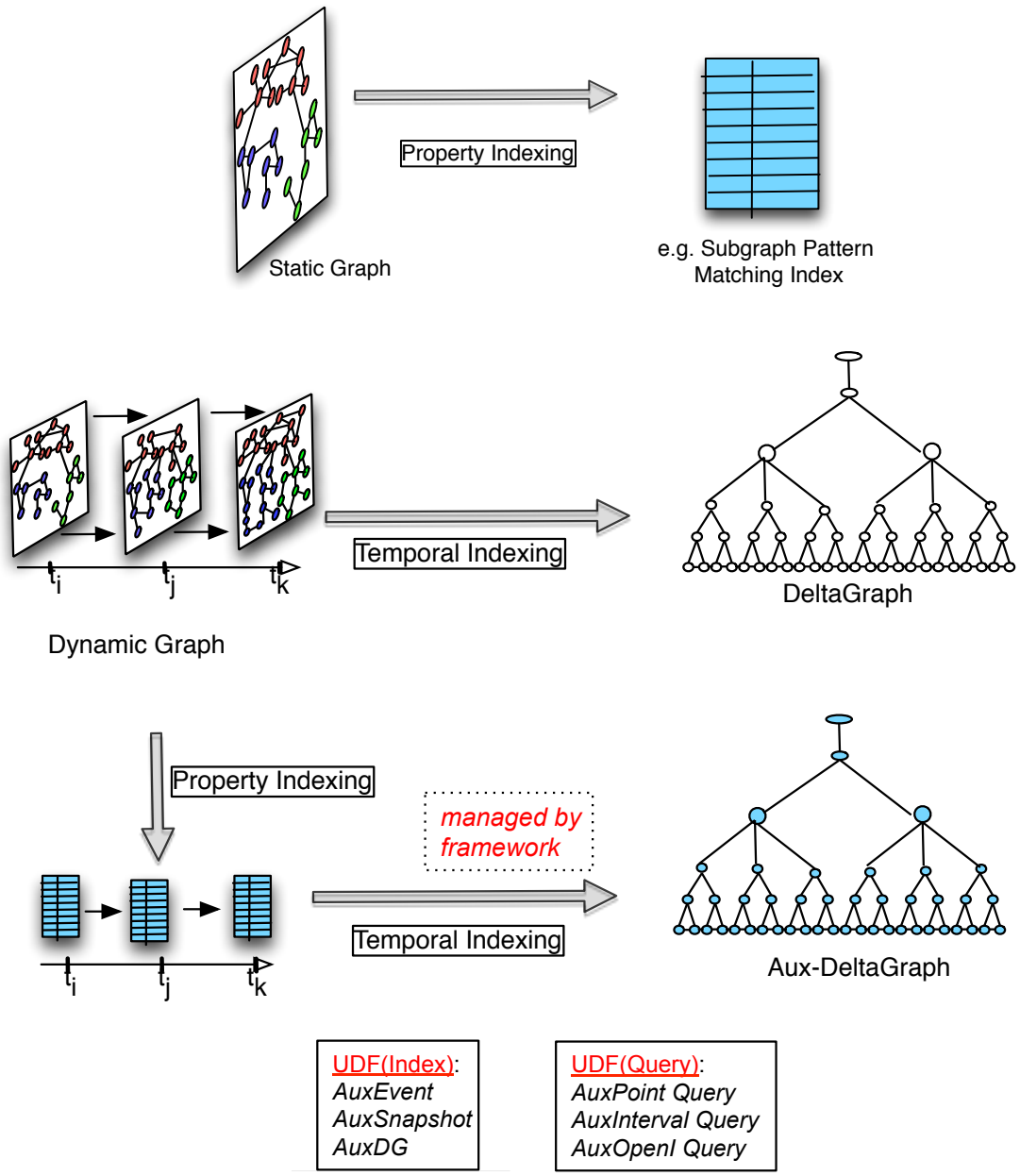


Figure 3.8: The *extensibility* framework of DeltaGraph is meant for users or programmers to execute specific historical queries efficiently by only supplying a logic through *user defined functions* while the framework takes care of the temporal aspects of the implementation.

We can extend the basic DeltaGraph structure to support such an index as follows. The auxiliary information maintained is precisely the pattern index at each snapshot, which is naturally stored in a compact fashion in the DeltaGraph (by exploiting the com-

monalities over time). It involves implementation of the AuxIndex and AuxHistQueryInterval classes accordingly. For example, the *CreateAuxEvent*, creates an AuxEvent, defining the addition or deletion of a path of four nodes, by finding the effect of a plain Event (in terms of paths) in the context of the current graph. Instead of using the standard differential function, we use one that achieves the following effect: a specific path containing 4 nodes, say $\langle n_1, n_2, n_3, n_4 \rangle$, is present in the pattern index associated with an interior node if and only if, it is present in all the snapshots below that interior node. This means that, if the path is associated with the root, it is present throughout the history of the network. Such auxiliary information can now be directly used to answer subgraph pattern matching query to find all matching instances over the history of the graph. We evaluated our implementation on Dataset 1 (details in the Section 3.7), and assigned labels to each node by randomly picking one from a list of ten labels. We built the index as described above (by indexing all paths of length 4). 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.

This extensibility framework (Figure 3.8) enables us to quickly design and deploy reasonable strategies for answering many different types of queries. We note that, for specific queries, it may be possible to design more efficient strategies. In ongoing work, we are investigating the issues in answering specific types of queries over historical graphs, including subgraph pattern matching, reachability, etc.

3.5 DeltaGraph Analysis

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

3.5.1 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.

3.5.2 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 3.2 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 $\frac{1}{2}\rho_{ab}$.

Table 3.2: A list of Differential Functions

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

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

The other functions shown in Table 3.2 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.

3.5.3 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 analysis on Balanced and Intersection differential functions.

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. Consider an interior node p with k children, $c_1, \ldots, c_k$. If p is at level 2 (i.e., if c_i's are leaves), then $S_p = S_{c_1} + \frac{1}{2}\delta_{c_1c_2} - \frac{1}{2}\rho_{c_1c_2} + \frac{1}{2}\delta_{c_2c_3} - \cdots$. It is easy to show that:

$$\begin{aligned} |\Delta(p, c_i)| &= \frac{1}{2}(|\delta_{c_1c_2}| + |\rho_{c_1c_2}| + \cdots), \quad \forall i \\ &= \frac{1}{2}(k-1)(\delta_* + \rho_*)L, \quad \forall i \end{aligned}$$

The number of edges between the nodes at the first and second levels is N, thus the total space required by the deltas at this level is: $N\frac{1}{2}(k-1)(\delta_* + \rho_*)L = \frac{1}{2}(k-1)(\delta_* + \rho_*)|E|$.

If p is an interior node at level 3, then the distance between c_1 and c_2 in terms of the number of events is exactly $k(\delta_* + \rho_*)L$. This is because c_2 contains: (1) the insert and delete events from c_1's children that c_1 does not contain, (2) $(\delta_* + \rho_*)$ events that occur between c_1's last child and c_2's first child, and (3) a further $\frac{(k-1)}{2}(\delta_* + \rho_*)L$ events from its own children. Using a similar reasoning to above, we can see that:

$$|\Delta(p, c_i)| = \tfrac{1}{2}(k-1)k(\delta_* + \rho_*)L, \quad \forall i$$

Surprisingly, because the number of edges at this level is $\frac{N}{k}$, the total space occupied by the deltas at this level is same as that at the first level, and this is true for the higher levels as well.

Thus, the total amount of space required to store all the deltas (excluding the one from the empty super-root node to the root) is:

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

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.

Finally, using the 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.

$\underline{\rho_* = 0}$: For a growing-only graph, root snapshot is exactly G_0.

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

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

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 $|G_0|$.

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).

3.5.4 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.

3.6 Construction and Maintenance

3.6.1 Batch Construction

Besides the graph itself, represented as a list of all events in a chronological order, E, the DeltaGraph construction algorithm accepts four other parameters: (1) L, the size of a leaf-level eventlist; (2) k, the arity of the graph; (3) $f()$, the differential function that com-

putes 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 deltas corresponding to the edges 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 have the option to use the GraphPool data structure (explained in Chapter 5) to maintain these snapshots in an overlaid fashion to decrease the total memory consumption. The savings of memory in that case come at the cost of a slight construction time hike. 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. DeltaGraph construction can be sped up using parallel processing through one of the two following techniques:

1. By dividing the dataset into multiple partitions by nodeid, as explained above, and processing each partition in parallel.

2. Besides or instead of the above, the set of events, E are divided into contiguous uniform chunks of event sets, maintaining the relative order of events. For each one of smaller event sets, compute the correct initial graph, G_0 corresponding to the

time of the first event. Using each G_0 and the event set, a DeltaGraph is constructed separately, in parallel. Upon completion, the DeltaGraphs are stitched together in the given order to form the overall DeltaGraph.

3.6.2 Configuring DeltaGraph

As discussed before, DeltaGraph is defined by the specification of different parameters - l, df, k. The user of the index can specify desired values at the time of the construction. A discussion of the effect of different parameter values can be found in Section 3.5.4 for general understanding of their behavior. The default differential function, $Intersection$, results in maximum temporal compression, hence smallest storage size and retrieval times. Only in the case of a specific requirement, other functions should be used, such as the *Balanced* function. Given the set of events, E, and a particular choice of the arity, k, we observed that the average time of snapshot retrieval depends on two different factors - the height (h), and the eventlist size (l) of the DeltaGraph. The magnitude of both the quantities is proportional to the average snapshot retrieval time. This can be justified as follows: bigger the eventlist sizes, higher the number of events an average query reads, hence higher the query time; higher the height of DeltaGraph, higher is the *number* of deltas read, and hence higher the query latency.

We model the average query latency in terms of the two parameters as follows, where a, b, c are empirically determined constants.

$$T_a = a.h + b.l + c$$

Using $h = \log_k \frac{|E|}{l} + 1 \approx \log_k \frac{|E|}{l}$,

$$T_a = a.\log_k \frac{|E|}{l} + b.l + c$$

Differentiating T with respect to l, assuming a differentiable T_a, and equating to 0, we get the following value of l,

$$l = \frac{a}{b \log k}$$

Given a k, l and E, optimal value l should be set as per the given formula. Here a, b, c, are constants that that account for machine hardware, network speed, underlying datastore, and the relative cost of reading an event compared to a node of the graph.

The value of k suggests the maximum arity or fan out for a DeltaGraph. Increasing the value of k for a fixed l and $|E|$ decreases the height, h, of the structure. And a lower height signifies lesser number of delta lookups for snapshot retrieval, which in turn would decrease the retrieval time. Empirically, it is seen that there is a noticeable dip in the retrieval times for $k = 3$ from $k = 2$, after which the benefit is much more gradual. However, the down side of a higher k value suggests lesser temporal abstraction and a higher amount of space required by the DeltaGraph. Consider for instance, $k \approx |E|/l$, suggests an arity as high as the total number of base snapshots. The only temporal abstraction that takes place is for the elements of the graph that were consistently present through out its time span. In the likely absence of a non-empty parent snapshot, it essentially reduces the DeltaGraph to a Copy+Log approach, which may be good for snapshot retrieval, but terrible for storage constraints. Given that there is only a marginal benefit in retrieval la-

tencies for arity values higher than 3 or 4, and a constant increase in space requirements, we recommend restricting the arity to $k = 3 \pm 1$. The supporting empirical evidence can be seen in Section 3.7.

3.6.3 Updates

There exist multiple valid DeltaGraphs for a single dataset and configuration. For a chronological set of events, E, a differential function, $f()$, eventlist size, l, and arity, k, we can obtain a variety of DeltaGraphs such as those illustrated in Figure 3.10.

3.6.3.1 Preliminaries

A *Packed DeltaGraph* has the following properties for a particular configuration and event set:

- At any level, only one node can have less than k children.
- All but one leaf nodes must be exactly l events apart.
- All nodes based on the default differential function.

A *Right Packed DeltaGraph* is a specific case of the packed DeltaGraph where the node with less than k children or the leaf node pair with less than l events in between are the rightmost one at their respective levels. There exists one and only one right packed DeltaGraph for a given configuration and input (the proof is trivial and hence omitted).

An α-*Packed DeltaGraph* is a:

- Right packed GeltaGraph with certain relaxations:

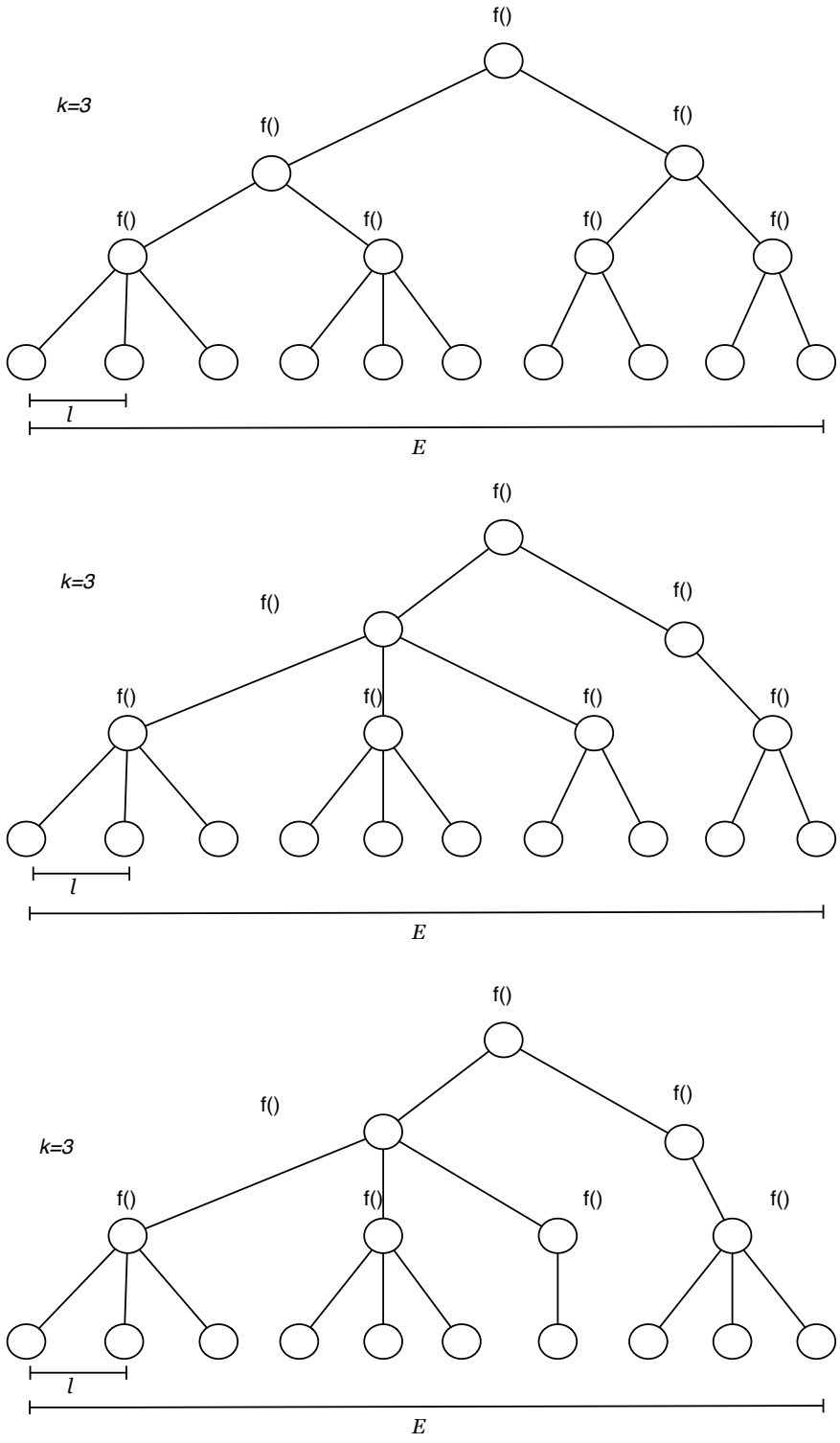


Figure 3.9: Different valid DeltaGraphs for input E, configurations, $df = f(); k = 3; l$. All of them have the same number of nodes at each corresponding level, but the structure is different.

- The node with less than k degree of children can have a non-default empty (ϕ) differential function

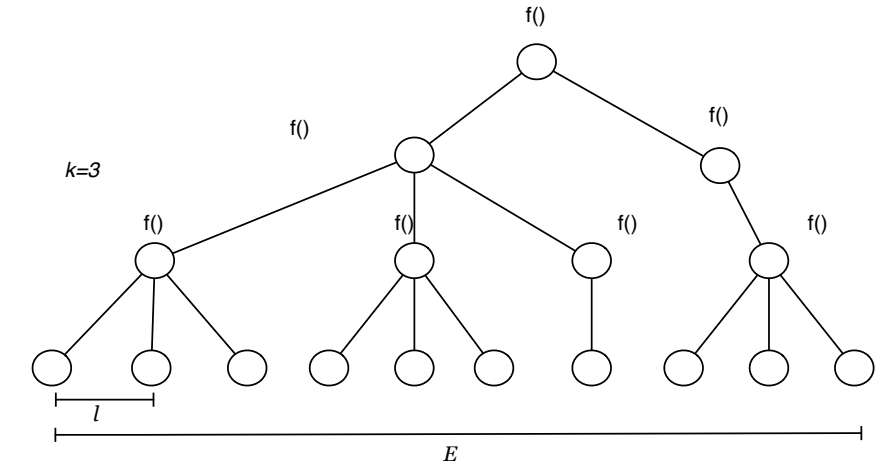
- The node with a child that has an empty differential function, can have an empty differential function itself.

The number of non-default empty differential functions is called the α-difference (α_D) of an α-Packed DeltaGraph. Note that there exists one and only one α-Packed DeltaGraph for one set of configurations. The β-difference for a level $r > 0$(not defined for leaf), $\beta_{r,D}$, is the minimum number of children its rightmost node is short of k. Finally, *Symmetry Deficiency*, S_D is the minimum number of events required for an α-Packed DeltaGraph to attain an α-difference of 0. $S_D = k^{\lfloor \log_k (\frac{|E|}{l}+1) \rfloor + 1} - E$.

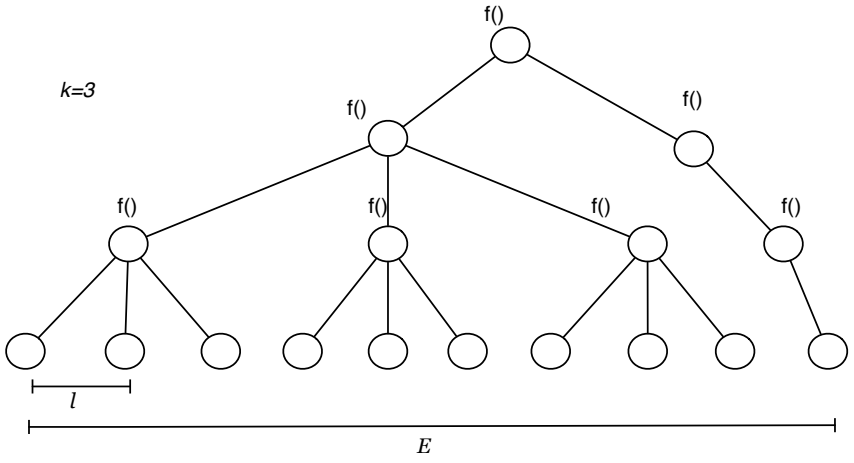
3.6.3.2 Update Procedure

Our strategy to update a DeltaGraph is based upon continuously maintaining an α-Packed DeltaGraph. Given an α-Packed DeltaGraph D_1 constructed upon E events, using a differential function, $df = f$, arity, k, and eventlist size, l; for new events E', such that they chronologically proceed E, we determine the α-Packed DeltaGraph for the set $E + E'$. When a new l events are pushed to an α-Packed DeltaGraph, it forms another snapshot (leaf node), and depending on the prior value of $\beta_{1,D}$, it is adjusted into the DeltaGraph. If previously,

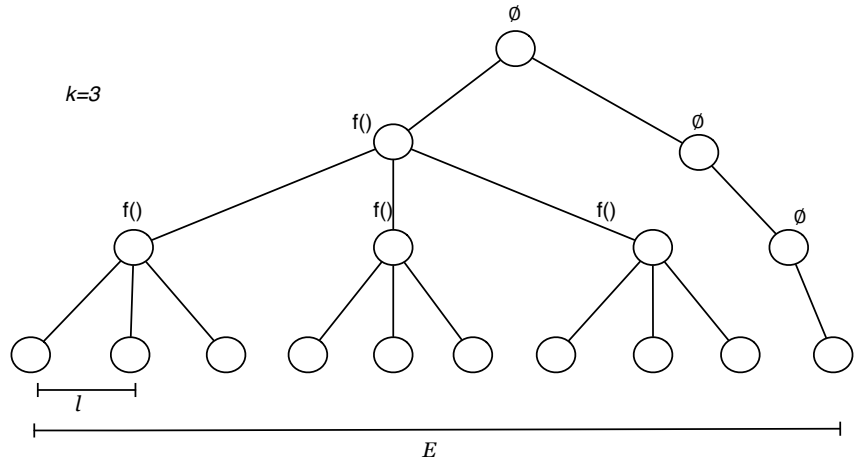
- $\beta_{1,D} = 0$, a new parent at $r = 1$ is created, and this is rolled up recursively for the next level;



(a) Packed DeltaGraph



(b) Right Packed DeltaGraph



(c) α-Packed DeltaGraph

Figure 3.10: Different Packed DeltaGraphs for input E, configurations, $df = f(); k = 3; l$.

- $\beta_{1,D} = 1$, the rightmost child at $r = 1$ is assigned as its parent and the differential function changed from $df = \phi$ to $df = f$, and deltas are changed appropriately;

- $1 < \beta_{1,D} < k$, the snapshot as is added as a new leaf node and the rightmost node at $r = 1$ is set as its parent;

In all cases, the following operation is performed at the end, $\beta_{r+1,D} = (\beta_{r+1,D} + 1)\%k$ for all levels r where a node has been added.

The procedure for continuously maintaining an updated α-Packed DeltaGraph works with adding events in batches of multiples of l. While the update is happening, all the new deltas from a parent with less than k children are preferably kept in memory because they are likely to be modified with the change of differential function to f. At the end of update, all deltas are flushed to disk. During an update, when a delta is flushed to disk, we give it an alias id, without overwriting any existing deltas. At the end of update, during a *lockout period*, the old delta-ids are assigned to the aliases, and the old deltas are lost. This ensures a minimal down time (lockout period) from the DeltaGraph index during update. The procedure trivially extends to non-multiples of l. However, due to reasons of efficiency, we recommend updating in the intervals of l, and storing the remaining events (less than l at any time) in memory. It can be seen that the work required to update a batch of events is always less than or equal to the update it in smaller batches. This is because a batch update reduces extraneous disk writing and reading. Finally, since the updates are coordinated centrally, we ensure data integrity in case of local update failure.

3.7 Experiments

In this section, 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 discussed in Section 3.3.3; in addition, we have implemented a Pregel-like iterative framework for distributed processing, and the subgraph pattern matching index presented in Section 3.4.5.

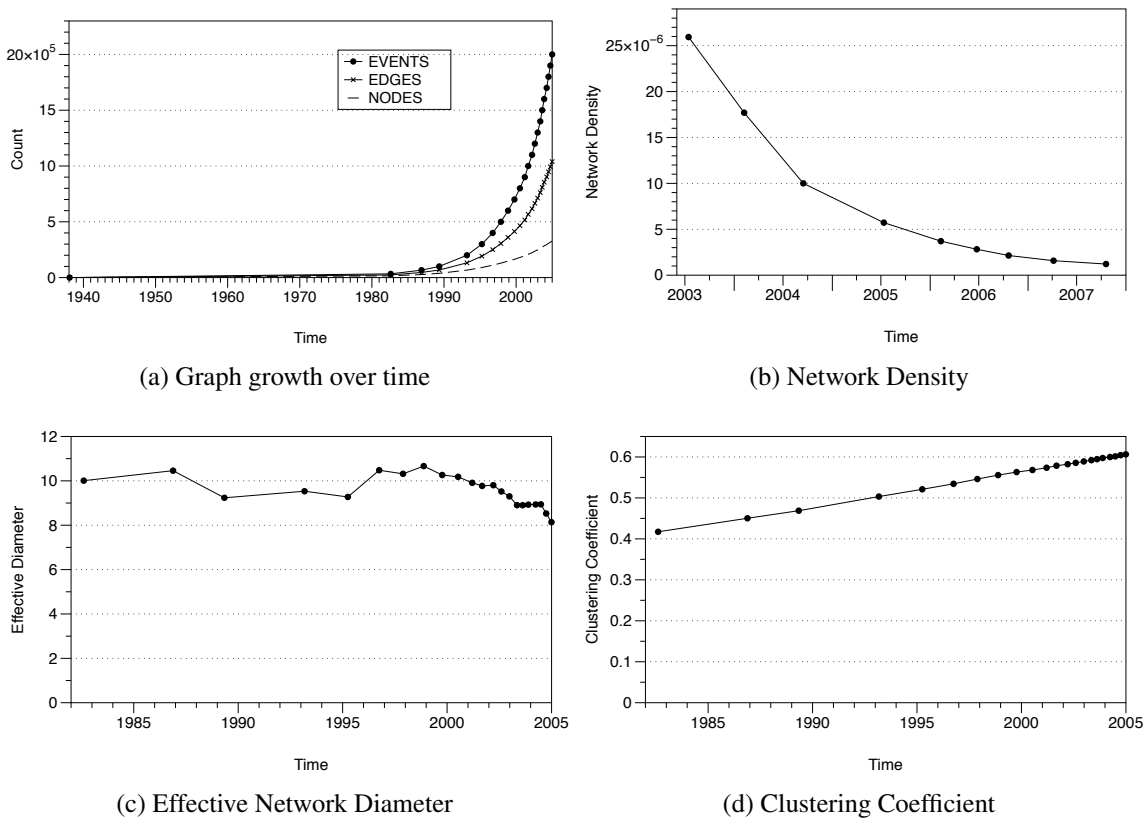
Datasets: We used four datasets in our experimental study.

(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. The growth of the network over time can be seen in Figure 3.11. The growth (node count and edge count) of the network is exponential. The density constantly decreases over time, while the clustering coefficient increases. The effective network diameter[2] decreases over time.

(2) Dataset 2 is a randomly generated historical trace with 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

[2] *Effective network diameter* is defined as the minimum diameter for at least 90% of the nodes [102].



(a) Graph growth over time

(b) Network Density

(c) Effective Network Diameter

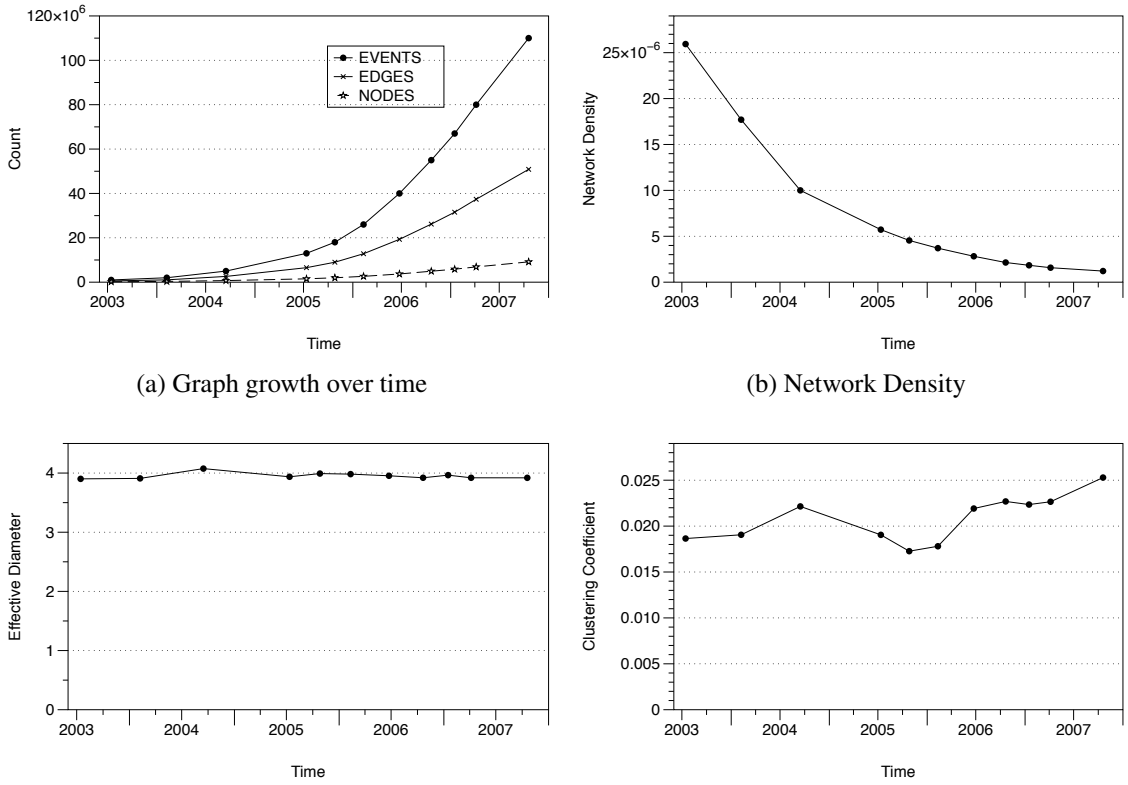
(d) Clustering Coefficient

Figure 3.11: Summary of Dataset 1 over time.

10 million ($10M$) edges and $3M$ nodes (from a patent citation network), followed by $100M$ events, $50M$ edge additions and $50M$ edge deletions.

(4) Dataset 4 is a Wikipedia citation network consisting of 110 million edge addition events from January 2001 to October 2007. Its characteristics over time can be seen in Figure 3.12. Compared to the the DBLP network (dataset 1), the the Wikipedia network exhibits a smaller diameter and a lesser value of clustering coefficient.

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



(a) Graph growth over time

(b) Network Density

(c) Effective Network Diameter

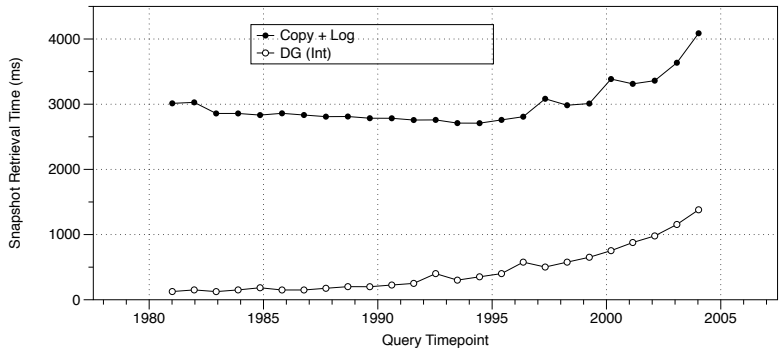
(d) Clustering Coefficient

Figure 3.12: Summary of Dataset 4 over time.

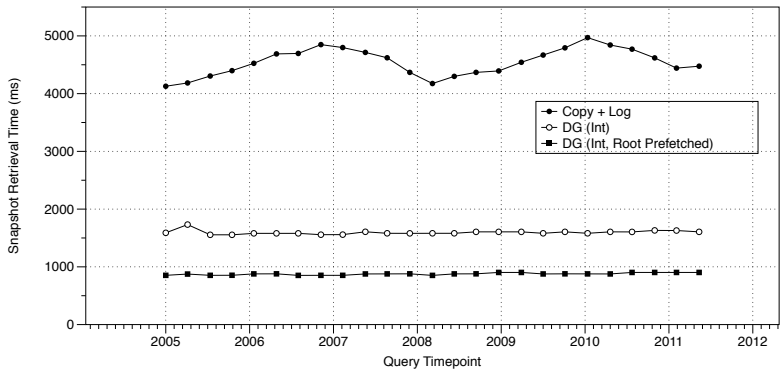
compression in Kyoto Cabinet). On average, it took us 23.8 seconds to calculate PageRank for a specific graph snapshot, including the snapshot retrieval time. This experiment illustrates the effectiveness of our framework at scalably handling large historical graphs.

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, the experiments were run on a single Amazon EC2 core (with 1.4GB memory).

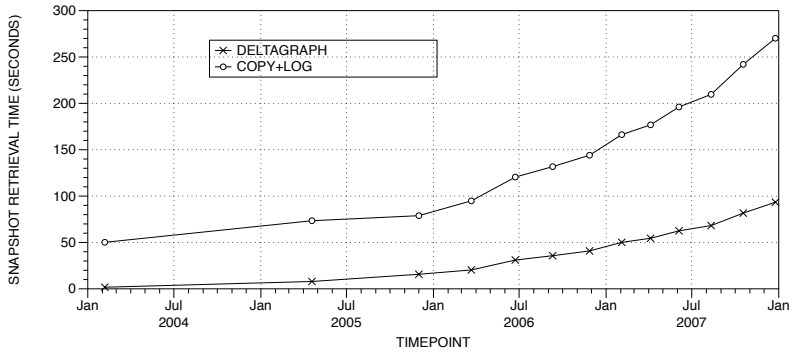
Comparison with other storage approaches: We begin with comparing our approach with **in-memory** interval trees, and Copy+Log approach.



(a) Performance: Dataset 1a



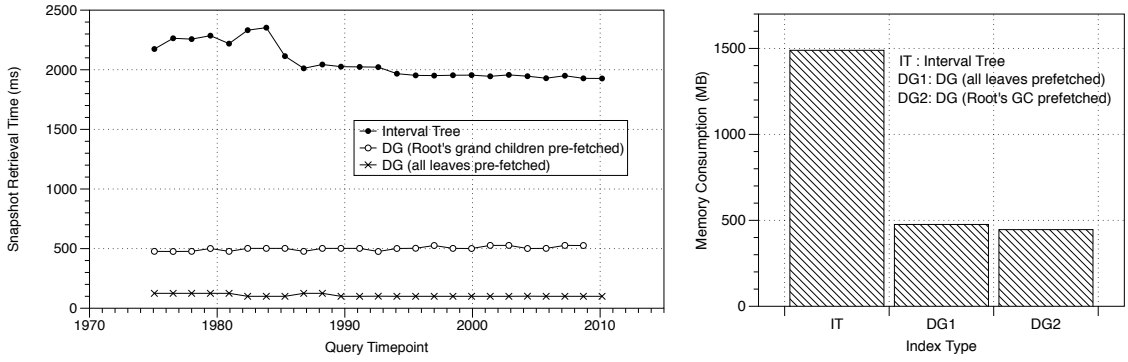
(b) Performance: Dataset 2



(c) Performance: Dataset 4

Figure 3.13: Comparing DeltaGraph and Copy+Log. *Int* and *Bal* denote the Intersection and Balanced functions, respectively.

Figure 3.13 shows the results of our comparison between Copy+Log and DeltaGraph approaches for time taken to execute 25 uniformly spaced queries on Datasets 1, 2 and 4. 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



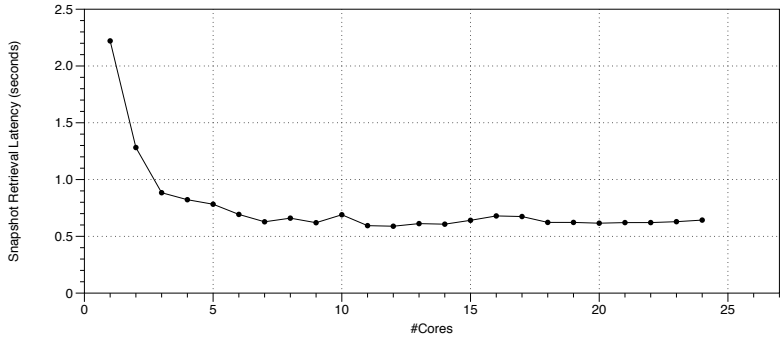
(a) Performance (b) Index Memeory

Figure 3.14: Performance of different DeltaGraph configurations vs. Interval Tree for Dataset 2.

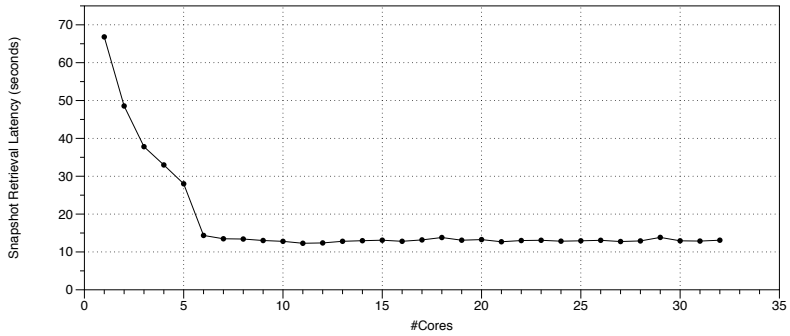
950MB for Dataset 1 and 2, respectively), the DeltaGraph could afford a smaller size of L and hence higher number of snapshots than the Copy+Log approach. As we can see, the best DeltaGraph variation outperformed the Copy+Log approach by a factor of at least 4, and orders of magnitude in several cases.

Figure 3.14 shows the comparison between 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 graphs). The average retrieval times were worse than DeltaGraph by factors of 20 and 23 for Datasets 1 and 2 respectively.



(a) Dataset 1

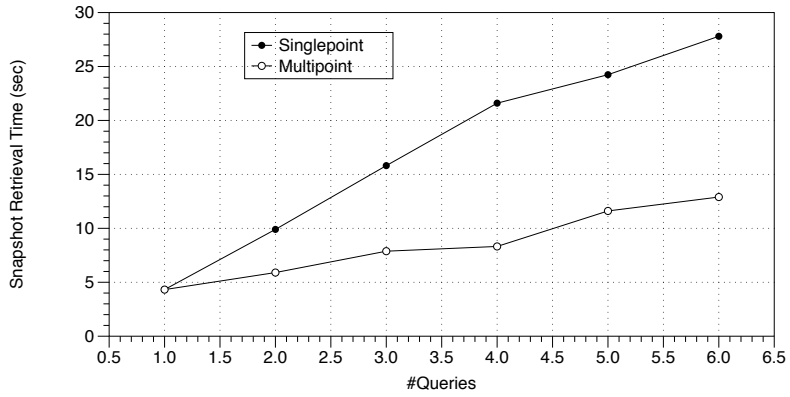


(b) Dataset 4

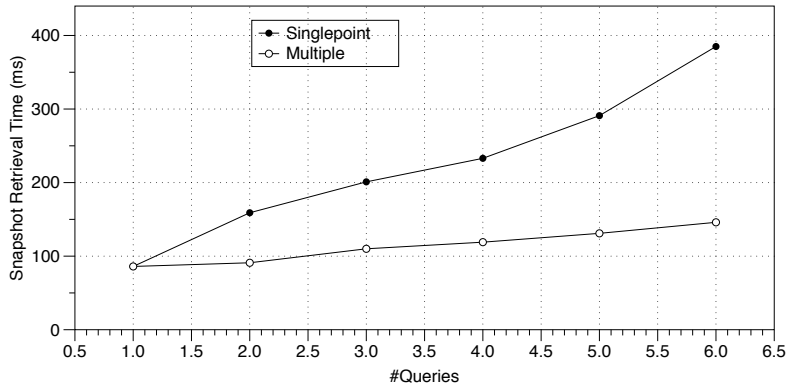
Figure 3.15: Benefits due to multi-core parallelism for snapshot retrieval on a single machine with single disk.

Multicore Parallelism: Figure 3.15 shows the advantage of concurrent query processing on a multi-core processor using a partitioned DeltaGraph approach, where we retrieve the graph in parallel using multiple threads. We observe high speedups up-to 4-6 cores on a single machine, after which the performance saturates. This further validating our parallel design.

Multipoint queries: Figure 3.16 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 and 4. The x-axis represents the number of snapshot queries, which were chosen to be 1 month apart. As we can see, the advantages of multi-point query retrieval are quite significant because of the high overlap between the retrieved



(a) Dataset 1



(b) Dataset 4

Figure 3.16: Multipoint query execution vs multiple singlepoint queries.

snapshots.

Advantages of columnar storage: Figure 3.17 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 by more than a factor of 3.

Effect of DeltaGraph construction parameters: We measured the average query times and storage space consumption for different values of the arity (k) and leaf-eventlist sizes (L) for Dataset 1. Figure 3.18 shows that with an increase in the arity of the DeltaGraph, the average query time decreases rapidly in the beginning, but quickly flattens. On the other hand, the space requirement increases in general with small exceptions when the

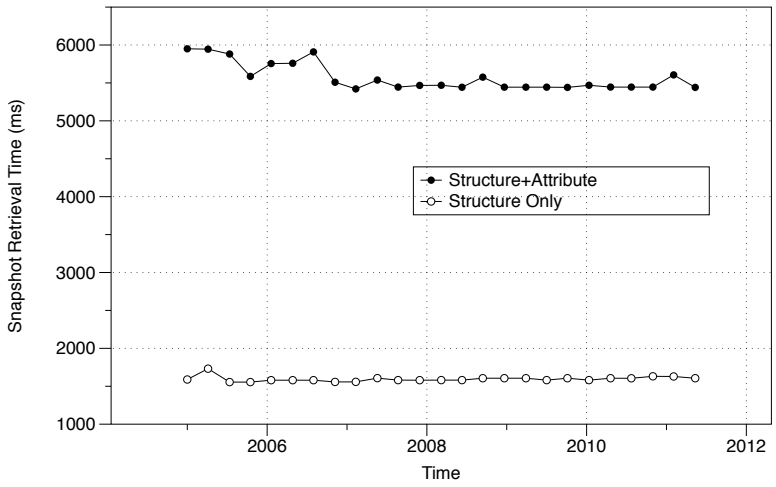


Figure 3.17: Retrieval with and without attributes (Dataset 2)

height of the tree does not change with increasing arity. We omit a detailed discussion on the exceptions as their impact is minimal. Again, referring back to the discussion in Section 3.5.4, the results corroborate our claim that higher arity leads to smaller DeltaGraph heights and hence smaller query times, but results in higher space requirements. The effect of the leaf-eventlist size is also as expected. As the leaf-eventlist size increases, the total space consumption decreases (since there are fewer leaves), but the query times also increase dramatically because of a linear increase in the average number of events to be read and processed. This can be seen in Figure 3.19.

Figure 3.21 shows the snapshot retrieval time for two DeltaGraphs on Dataset 2. One has a uniform eventlist size ($15,000$), while the other has the eventlist size as $30,000$ and $7,000$ for the first and the second half of the query time period, respectively. The difference in performance for the the two regions of the second DeltaGraph can be seen as a direct effect of the choice of eventlist size.

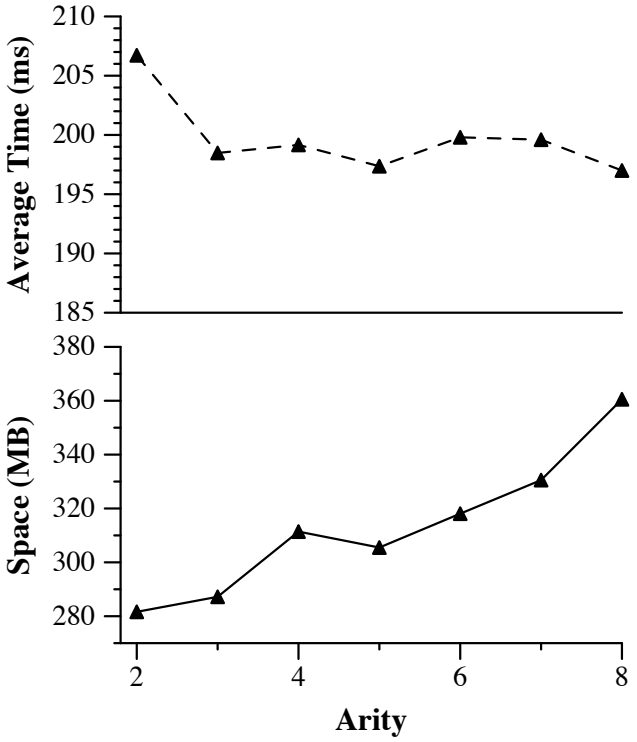


Figure 3.18: Effect of varying arity (Dataset 1)

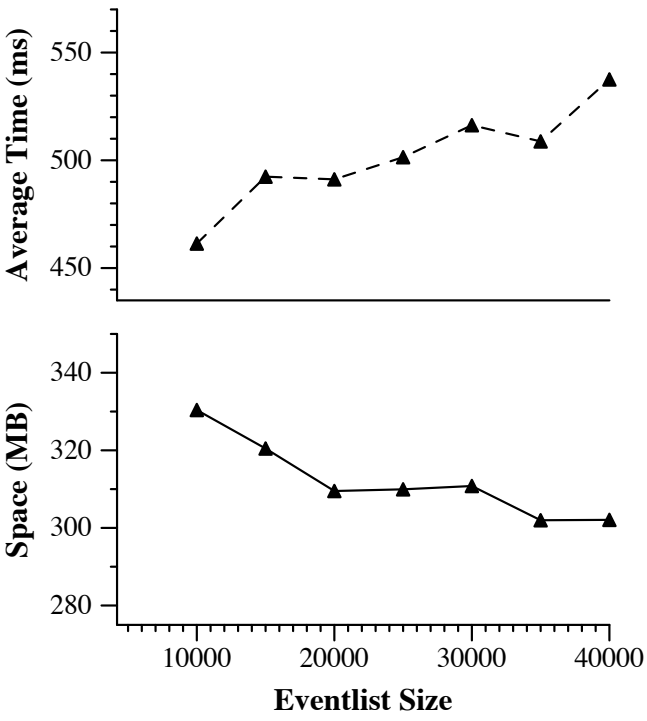


Figure 3.19: Effect of varying eventlist sizes (Dataset 1)

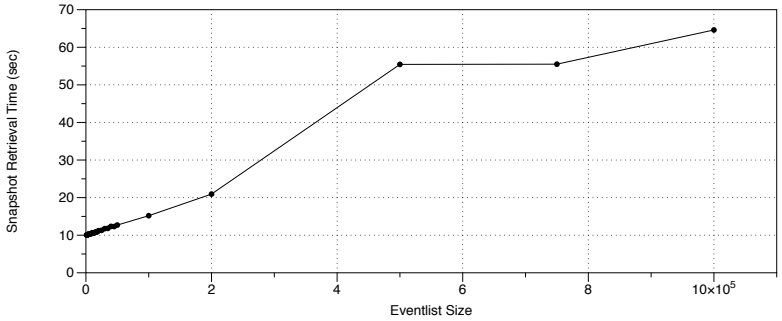


Figure 3.20: Effect of varying eventlist sizes on snapshot retrieval times (Dataset 4)

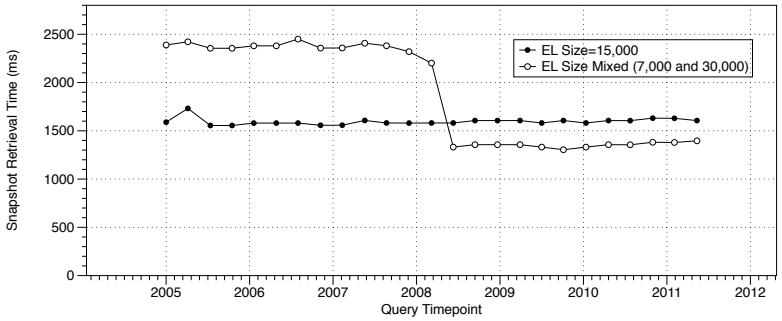
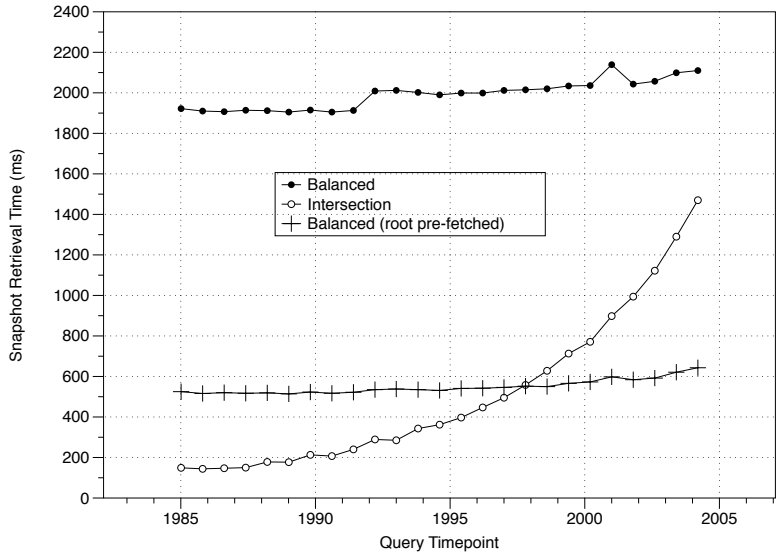


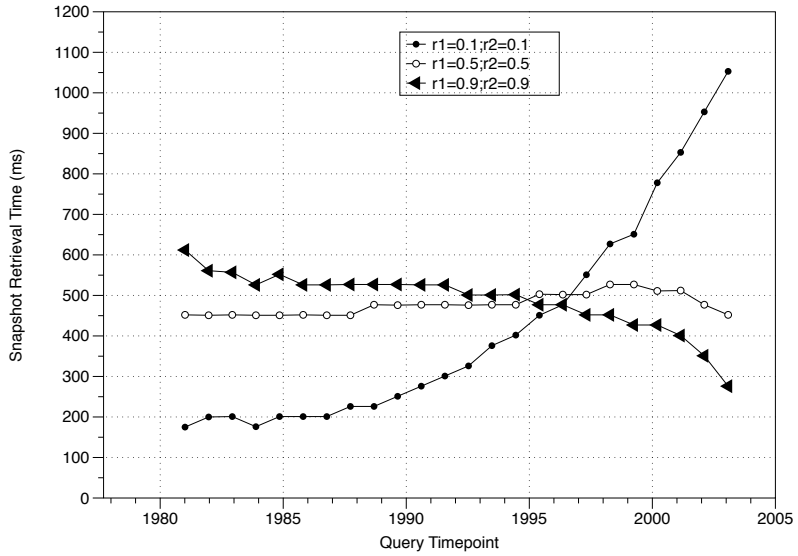
Figure 3.21: Snapshot retrieval performance on a DeltaGraph with different eventlist sizes for different time periods (Dataset 2).

Differential functions: In Section 3.5.4, we discussed how the choice of an appropriate differential function can help achieve desired distributions of retrieval times for a given network. Figure 3.22(a) compares the behavior of Intersection and Balanced functions with and without root materialization on Dataset 1. On the growing-only graph, using the Intersection function results in skewed query times, with the larger (newer) snapshots taking longer to load.

The balanced function, on the other hand, provides a more uniform access pattern, although the average time taken is higher. By materializing the root node, the average becomes comparable to that of Intersection, yielding a uniform access pattern. A few different configurations for the mixed function are shown in Figure 3.22(b), with $r_1 = 0.5$, $r_2 = 0.5$ denoting the Balanced function. As we can see, by choosing an appropriate



(a)



(b)

Figure 3.22: Comparison of snapshot retrieval performance for differential functions on Dataset 1. (a) *Intersection* and *Balanced*; (b) Different *Mixed* function configurations.

differential function, we can exercise fine-grained control over the query retrieval times thus validating one of our main goals with the DeltaGraph approach.

Backend Storage While we have reported all experiments so far using Kyotocabinet as the backend key-value store for the deltas, the system implementation is easily customiz-

able to use any other key-value store, such as Cassandra or HBase. Our choice to employ Kyotocabinet in this prototype was based upon the fact that provides really fast read and write performance on local disk. Figure 3.23, shows the snapshot retrieval query latencies for Kyotocabinet and Cassandra on a small sample of queries on Dataset 4. We used a single machine setup with local databases for both. Cassandra was used without replication. Kyotocabinet supports slightly faster reconstruction process. Cassandra, however comes with other benefits, which we shall also explore in Chapter 3.

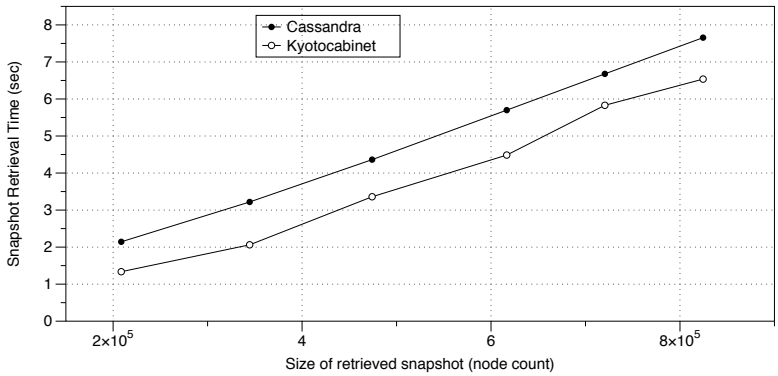


Figure 3.23: DeltaGraph snapshot retrieval using single machine, local configurations of Kyotocabinet and Cassandra (Dataset 4).

3.8 Conclusion

In this chapter, we addressed the problem of compactly storing historical data for large information networks and optimally reconstructing past snapshots from it. We presented DeltaGraph, a distributed hierarchical index structure that exploits temporal redundancy to store the history of a network in form of snapshot differences, called deltas. It is a flexible and highly tunable data structure through which we can trade decreased snapshot retrieval times for increased disk storage requirements. Our experimental evaluation

shows that the choice of DeltaGraph is superior to the existing alternatives. We showed both analytically and empirically that the flexibility of DeltaGraph helps control the distribution of query access times through appropriate parameter choices at construction time, and snapshot prefetching at runtime. Our experimental evaluation demonstrated the impact of many of our optimizations, including multi-query retrieval and columnar storage. DeltaGraph solves a core problem in historical graph indexing and sets up the stage for addressing further retrieval objectives such as version retrieval, which are presented in Chapter 4.

Chapter 4: Generalized Historical Graph Storage in the Cloud

4.1 Introduction

In Chapter 3, we studied the problem of compact indexing of historical graphs with the objective of efficient snapshot retrieval using DeltaGraph. Reconstructing past states of a historical graph is perhaps the single most important primitive task required for a historical graph index. However, snapshot retrieval is inefficient in fetching version-centric primitives. For example, in order to analyze the the history of a node or neighborhood, we require all the changes that occurred to a particular node or a subgraph. Recall that, in snapshot retrieval, we are given specific time point(s) at which all the components of the graph are required to be fetched. On the other hand, for retrieval of versions or history, we are given the node(s) to fetch, but the specific time points at which to fetch them, are not a part of the query specification (although the query may consist of a constraint on the bounding time interval). Hence, version retrieval calls for a fetch task of a inherently different nature than snapshot retrieval.

The types of retrieval queries of interest in temporal graph analytics can be summarized into three categorizes: (a) fetching one or more historical snapshots of the graph as of specified time points in the past, (b) retrieving all the changes to a single node over the lifetime of the node, and (c) retrieving the evolution history of a specified neighborhood

or connected subgraph over a specified period of time. In order to support all types of temporal analysis tasks over graphs, we need an index that can perform both, snapshot and version style of retrieval efficiently, over different granularities - from a single node to the entire graph. The two fundamentally different approaches to store and index time-evolving data can be classified as, one by snapshot indexing (time-centric) and the other by changes to individual items (entity-centric). These two represent two extreme design points, with the former being more suitable for snapshot retrieval queries, and the latter being more suitable for single-node evolution queries (neither supports the third type of query well). One of our motivations behind a new data store for historical graphs is to generalize these two notions into a unified, tunable, and scalable index.

In this chapter, we present *Temporal Graph Index (TGI)*, a highly scalable and elastic data store, that compactly indexes the entire history of a graph. 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 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 persist the partitioned deltas, and can thus leverage the scalability afforded by those systems (our implementation uses Apache Cassandra key-value store). TGI is a tunable index structure, and we investigate the impact of tuning the different parameters through an extensive empirical evaluation. Since TGI builds upon DeltaGraph, which is optimized for retrieving individual snapshots efficiently, we discuss the differences between the two in more detail in Section 4.2.

Data model: In the chapter, we employ a different data model for evolving graphs than the one used in the dissertation so far. So far we have considered that under a discreet notion of time, a time-evolving graph $G^{\mathcal{T}} = (V^{\mathcal{T}}, E^{\mathcal{T}})$ 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, 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.

In this chapter, our approach to temporal processing 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. A node contains its property attributes (key-values), as well as the property attributes of the edges that are its attributes in the first place. These attributes are temporal and change over time. It can be seen that a physical translation of this data model has to handle redundant information in form of edges which, although structurally fundamental to graph structure, are reduced to a secondary element here. While such modeling may seem counter-intuitive and counter-productive at first, it best mirrors the basis of storage and retrieval in TGI, as well as further processing of temporal graphs which (discussed in Chapter 5). The equivalence between this data model and the one described in Section 1.4 can be established trivially.

Background and Outline: We refer the reader to the prior work in graph indexing discussed in Section 2.3.3 and an account of snapshot retrieval in databases in Section 3.2.

The rest of this chapter is organized as follows. In Section 4.2, we describe the details of the TGI design, construction and graph fetching; we provide empirical efficacy of TGI in Section 4.3, followed by a summary in Section 4.4.

4.2 Temporal Graph Index

As we discussed perviously in Section 1.2, it is fairly inefficient to retrieve either snapshots or versions of specific nodes on a Log-based representation, because it requires reading a large volume of information even if the result is small. More efficient indexes on time evolving datasets can be based along either of the two principle dimensions – time or entity. Time based indexing methods such as *DeltaGraph* or *Copy+Log*, provide fast access to the graph at specific time points, i.e., snapshots. On the other hand, entity-based indexing approaches, e.g., a *node-centric temporal index*, would provide direct access to different versions of a specific node, but are fairly inefficient at fetching snapshots or subgraphs.

In this section, we investigate the issue of indexing temporal graphs and describe a *delta arithmetic* to define any temporal index as a set of different changes or *deltas*. 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 also describe the various access methods for graph primitives and compare the costs with alternatives using the delta arithmetic.

4.2.1 Preliminaries

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

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 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.

Definition 3 (Cardinality and Size)**.** The cardinality and the size of a Δ 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: $\Delta_1 + \Delta_2$ is not always equal to $\Delta_2 + \Delta_1$. We also note that: $\Delta_1 + \emptyset = \Delta_1$, and $(\Delta_1 + \Delta_2) + \Delta_3 = \Delta_1 + (\Delta_2 + \Delta_3)$.

Definition 5 (Δ Difference)**.** A 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 6 (Δ Intersection)**.** An intersection of two Δs is defined as a set intersection over the the components of two deltas. $\Delta_1 \cap \phi = \phi$, is true for any delta..

Definition 7 (Δ Union)**.** A 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 Δs:

Example 1 (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 Δ, 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., $\Delta_{event}(c, t_e) = c(t_e) - c(t_e - 1)$.

Example 2 (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.

Example 3 (Partitioned Eventlist)**.** An *partitioned eventlist* delta is an eventlist constrained by the scope of a set of nodes (say a set of nodes, $\mathcal{N} = \{N_1, N_2, \dots\}$) apart from the time range constraint $(t_s, t_e]$.

Example 4 (Snapshot)**.** A snapshot, $\mathcal{G}^{t_a}$ is the state of a graph $\mathcal{G}$ at a time point t_a. As a Δ, it is defined as the difference of the state of the graph at t_a from an empty set, $\Delta_{snapshot}(\mathcal{G}, t_a) = G(t_a) - G(-\infty)$.

Example 5 (Partitioned Snapshot)**.** A partitioned snapshot is a subset of a snapshot. It is identified by a subset of all nodes, $\mathcal{P}$ in graph, $\mathcal{G}$ at time, t_a. It consists of the state of all nodes at time t_a and all the edges whose at least one of the end points lies in $\mathcal{P}$ at time, t_a.

4.2.2 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.

Log: The Log index is equivalent to a set of all *event* deltas (equivalently, a single *eventlist* delta encompassing the entire history).

Copy+Log: 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 evolution between successive snapshots.

Vertex-centric: Although we are not aware of a specific proposal for vertex-centric index, a natural approach here would be to maintain a set of *partitioned eventlist* deltas, one for each node (with edge information replicated with the endpoints).

DeltaGraph: 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

Queries $\downarrow$	**Index** $\rightarrow$ **Size** $\rightarrow$	Log $\|G\|$	Copy $\|G\|^2$	C+L $\frac{\|G\|^2}{\|E\|}$	NC $2\|G\|$	DeltaGraph X_1^*	TGI X_2^{**}
Snapshot	$\sum_\Delta \|\Delta\|$	$\|G\|$	$\|S\|$	$\|S\|+\|E\|$	$2.\|G\|$	$h.\|S\|+\|E\|$	$h.\|S\|+\|E\|$
	$\sum_\Delta 1$	$\frac{\|G\|}{\|E\|}$	1	2	N	$2h$	$2h$
Static vertex	$\sum_\Delta \|\Delta\|$	$\|G\|$	$\|S\|$	$\|S\|+\|E\|$	$\|C\|$	$h.\|S\|+\|E\|$	$\frac{h.\|S\|}{p}+\frac{\|E\|}{p}$
	$\sum_\Delta 1$	$\frac{\|G\|}{\|E\|}$	1	2	1	$2h$	$2h$
Vertex versions	$\sum_\Delta \|\Delta\|$	$\|G\|$	$\|S\|\|G\|$	$\|G\|$	$\|C\|$	$\|G\|$	$\|V\|(1+\frac{\|S\|}{p})$
	$\sum_\Delta 1$	$\frac{\|G\|}{\|E\|}$	$\|G\|$	$\frac{\|G\|}{\|E\|}$	1	$\frac{\|G\|}{\|E\|}$	$\|V\|+1$
1-hop	$\sum_\Delta \|\Delta\|$	$\|G\|$	$\|S\|$	$\|S\|+\|E\|$	$\|R\|.\|V\|$	$h.(\|S\|+\|E\|)$	$\frac{h.(\|S\|+\|E\|)}{p}$
	$\sum_\Delta 1$	$\frac{\|G\|}{\|E\|}$	1	2	R	$2h$	$2h$
1-hop versions	$\sum_\Delta \|\Delta\|$	$\|G\|$	$\|S\|\|G\|$	$\|G\|$	$\|R\|.\|V\|$	$\|G\|$	$\|V\|(1+\frac{\|S\|}{p})$
	$\sum_\Delta 1$	$\frac{\|G\|}{\|E\|}$	$\|G\|$	$\frac{\|G\|}{\|E\|}$	$\|R\|$	$\frac{\|G\|}{\|E\|}$	$\|V\|+1$

Table 4.1: Comparison of access costs for different retrieval queries and index storage on various temporal indexes. In the table, C+L refers to Copy+Log and NC to Node-centric indexes. $|G|$ =number of changes in the graph; $|S|$ =size of a snapshot; h = height and $|E|$ = eventlist size in C+L, DG or TGI; $|V|$ =number of changes to a node; $|R|$=numbers of neighbors of a node; p= number of partitions in TGI. The metrics used are the sum of delta cardinalities ($\sum_\Delta |\Delta|$) and the number of deltas ($\sum_\Delta 1$). Finally, $^*X_1 = |G|(h+1)$; $^{**}X_2 = |G|(2h+3)$.

the intersection to the original snapshots, and recursively apply this till you are left with a single delta.

Table 4.1 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 the total storage space, which varies considerably across the techniques.

4.2.3 TGI Description

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

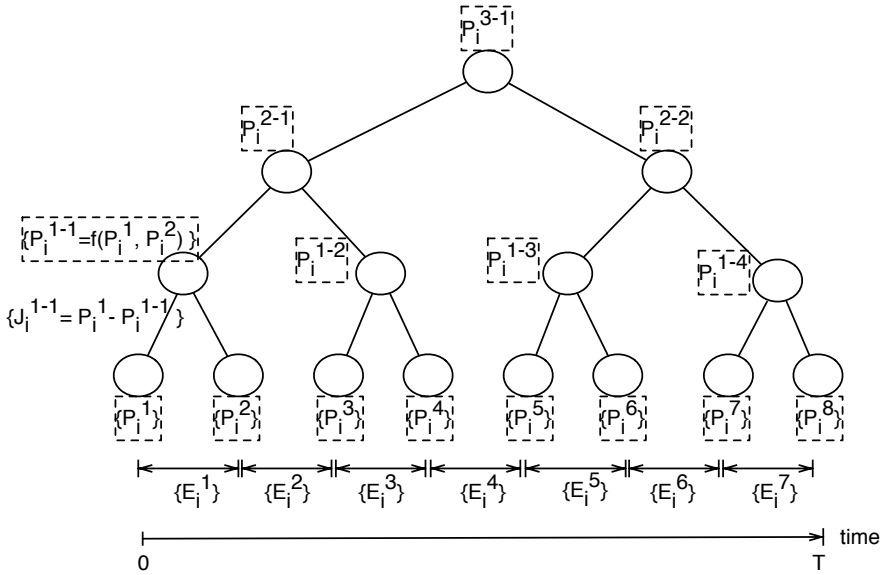
(a) Partitioned Eventlists: A set of partitioned eventlist Δs, $\{E_{tp}\}$, where E_{tp} captures the changes during the time interval t belonging to partition p.

(b) Partitioned Snapshots: 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 Δs, P^i_j, $1 < j < l$, such that $\cup_j P^i_j = \mathcal{G}^{t_i}$. There exists a function that maps any node-id(I) in $\mathcal{G}^{t_i}$ to a unique partition-id(P^i_j), $f_i : I \rightarrow P^i_j$. In practice, these snapshot deltas are not stored in their precise form, but in a hierarchical compressed form as described below in (c).

(c) Derived Partitioned Snapshots: Given a set of partitioned snapshots, we create a hierarchical structure similar to the DeltaGraph. Based on the partitioned snapshots, create parent snapshots per partition, using an *intersection* function. Further, we compute the difference between a snapshot and its parent, and call it the derived partitioned snapshot deltas. These are the deltas that we actually store in the index.

(d) Version Chain: For all nodes $\mathcal{N}$ in the graph $\mathcal{G}$, 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*(VC_I).

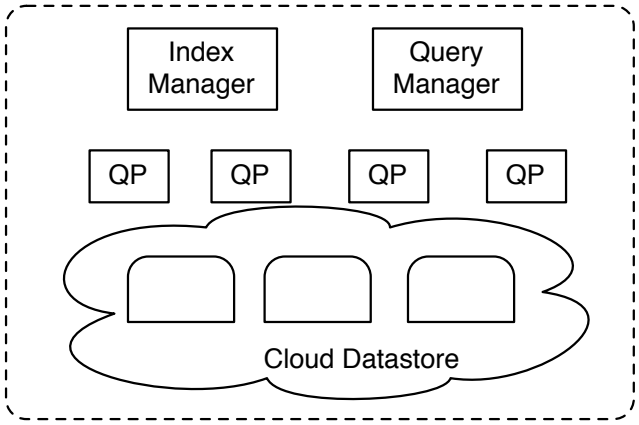
In short, the TGI stores *deltas* or *changes* in different forms, with a degree of repetition. The first one is the changes in a chronological order, organized in clusters through



(a) TGI deltas - partitioned eventlists, snapshots and derived snapshots. The (dotted) bounded deltas are not stored.

NODE	Version Chain
N1	$E_5^2 P_5^{1-2} E_5^6 ...$
N2	$E_1^2 E_2^2 E_3^2 ...$
N3	...
N4	...

(b) Version Chains



(c) Architecture

Figure 4.1: Temporal Graph Index representation.

graph partitioning. This facilitates direct access to the changes happening to a cluster or the graph as a whole. Secondly, the state of a cluster of nodes at different points in time is stored indirectly in form of the derived snapshot deltas. This facilitates direct access to the state of a cluster 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 somewhat direct access to the changes to individual nodes. Figure 4.1a shows the arrangement of eventlist, snapshot and derived snapshot partitioned deltas. Figure 4.1b shows a sample version chain.

4.2.4 Scalable TGI design

So far, we have explained the core ideas behind TGI – exploiting temporal consistency using a hierarchical index of finely partitioned snapshots, finely partitioned eventlists and a map for tracking the positions of nodes' changes, chronologically. We now discuss the key design and implementation aspects for scaling up TGI through cloud storage:

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. This is illustrated in Figure 4.2.

2. A graph at any point is horizontally partitioned into a fixed number (throughout graph's history) 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.

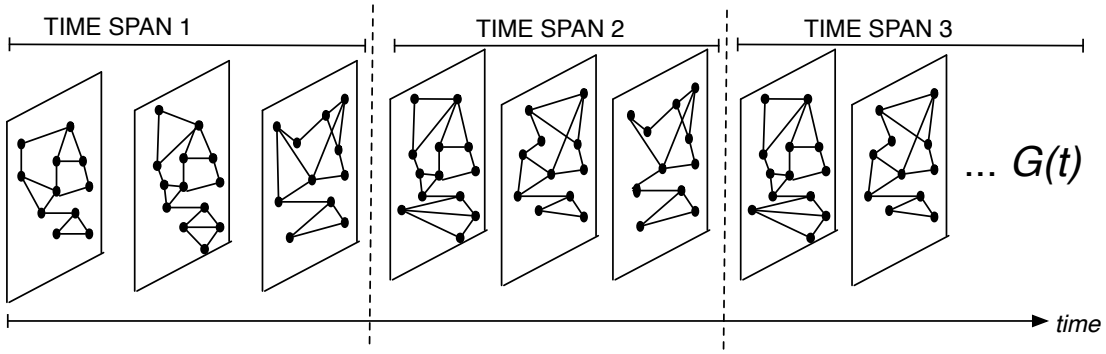


Figure 4.2: The graph history is divided into non-overlapping periods of time. Such division is based on time intervals after which the locality-based graph partitioning is updated. It is also used as a partial key for data chunking and placement.

3. The micro-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 micro-delta.

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. The micro-deltas are clustered by the delta key. The given order of the delta-key besides the placement-key elements, means that all the micro-partitions of a delta are stored contiguously, which makes it efficient to scan and read all micro-partitions belonging to a delta in a snapshot query. On the other hand, if the order of did and pid is reversed, it makes fetching a micro-partition across different deltas more efficient.

Irrespective of a temporal or a topological skew in the graph, the index is spread out across a cluster in a balanced manner. This also makes it possible to fetch the graph primitives of large sizes in a naturally parallel manner. For instance, a snapshot query would demand all micro-partitions for a specific set of deltas in a particular timespan across all horizontal partitions. Given an equitable distribution of the deltas across all machines of a cluster, we retrieve the data in parallel on each storage machine, without a considerable skew.

Implementation: TGI uses Cassandra [82] for its delta storage. There are 5 tables that contain TGI data and metadata:

(1) `Deltas(tsid, sid, did, pid, dval)` table stores the deltas as described above, where `dval` contains serialized value of the micro-delta as a binary string.

(2) `Versions(nid, vchain)` consists of each node's version chain as a hash-table with keys for each timespan.

(3) `Timespans(tsid, start, end, checkpts, k, df)` stores, for each timespan, start and end times, a list of snapshot checkpoints, and arity.

(4) `Graph(start, end, events, tscount, gtype)` contains global information about the graph and TGI.

(5) `Micropartitions(nid, tsid, pid)` contains micro-delta partitioning information about nodes. It is not utilized in case of random partitioning.

The graph construction and fetching modules are written in Python, using Pickle and Twisted libraries for serialization and communication.

Architecture: TGI architecture can be seen in Figure 4.1c, where *Query Mananger (QM)* is responsible for planning, dividing and delegating the query to one or more *Query Processors (QP)*. The 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. The *Index Manager* is responsible for the construction and maintenance activities of the index. The cloud represents the distributed datastore.

Construction and Update: The construction process involves three different stages. First, we analyze the input data using the index construction parameters including the timespan length (ts), number of horizontal partitions (ns), number of likely datastore nodes (m), eventlist size (l), and micro-delta partition size (psize). In the second stage, the input data is split into horizontal partitions. In the third stage, parallel construction workers of the IM work on separate horizontal partitions, and build the index, a time span at a time. The process of construction of each timespan is similar to that of DeltaGraph, albeit more fine-grained due to delta partitioning and version chain construction as well. The TGI accepts updates of events in batches of timespan length. The update process involves creating an independent TGI with the new events, and merging it with the original TGI. The merger of TGIs involves updates of corresponding deltas, VC index and the metadata. We refer the reader to our prior work details on DeltaGraph construction and updates to Section 3.6.

4.2.5 Partitioning and Replication

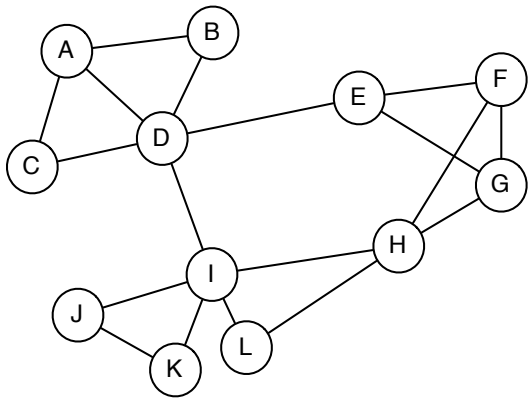
The key to scalability of TGI is partitioning, and it employs several types of partitioning that are orthogonal to each other. These include horizontal partitioning by nodes (performed at two different stages during construction), temporal partitioning across time, and columnar partitioning within nodes to store different attributes separately. The principle reason for partitioning into small, fine-grained deltas is to speed up the subgraph or node-centric retrieval queries. The original DeltaGraph design [73] used bigger deltas, which lead to wasteful I/O for such queries that target a small portion of the graph (as reflected in Table 4.1. Smaller delta sizes significantly improve the performance of such queries. On the other hand, smaller delta sizes reduce the performance for snapshot retrieval queries, both because of a larger number of queries required and the additional effort in putting the deltas together into the snapshot. The different construction parameters of TGI can be used to control the performance of one class of queries at the expense of the other class of queries.

TGI also performs replication at several different places (on top of the replication done by the distributed datastore for persistence). First, information about an edge is replicated in two different deltas if its endpoints are split. In addition, we may choose to replicate node information to speed up node or neighborhood retrieval queries. More specifically, consider a 1-hop neighborhood retrieval query for a given node u. If the nodes are randomly partitioned across the deltas, then fetching each of the neighbors of the node u is likely to require a separate query to fetch a separate delta. We address this problem through two mechanisms. First, we allow using a graph partitioning algorithm

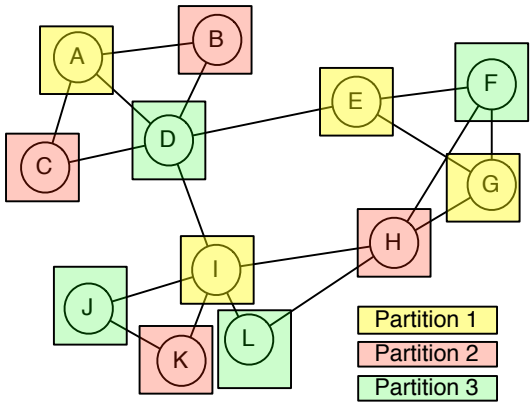
to create the partitions rather than creating them randomly (i.e., using a hash function). Although the problem of finding optimal k-way partitioning of a graph is NP-complete, several efficient heuristics are known that can scale to large graph sizes. The disadvantage of this approach is that we need to maintain a node-to-partition mapping explicitly to keep track of where a node is located.

Second, we replicate the data of the so-called *boundary* nodes that contain neighbors in other partitions, into those partitions [107]. Such replication helps speedup 1-hop replication costs as now each node's 1-hop neighborhood is contained in exactly one micro-delta. However, it affects the performance of snapshot retrieval and single node based retrieval. In order to avoid such extra penalties and still be able to perform fast 1-hop retrieval, we use the following strategy. The entire replicated portion for a micro-delta (i.e., parts that have been replicated *from* other micro-deltas) is stored as a separate micro-delta, called as an *auxiliary micro-delta*. It simply means that each (static) 1-hop retrieval now requires two micro-delta look-ups and there is no negative impact on the performance of snapshot or node retrieval. This is illustrated in Figure 4.3.

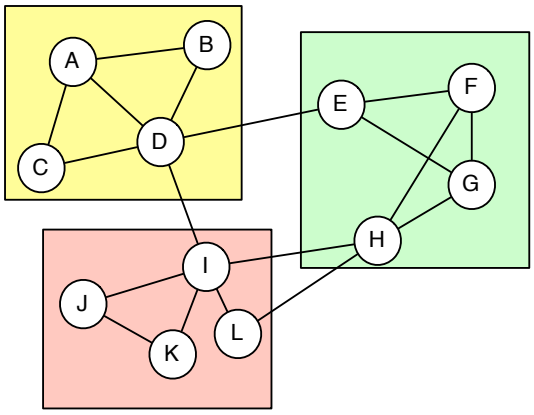
The amount of replication needed to ensure that all neighborhoods are fully contained within some partition unfortunately tends to be quite high. We partitioned 3 different graphs using Metis for the same partition size, $\frac{N}{k} = 1000$, a road network, DBLP co-authorship network and Orkut friends network; those three required 8%, 60% and 352% replication to ensure that every node's 1-hop neighborhood is located in in the same partition as the node. To combat this, we also allow partial replication, e.g., replicating edgelists of neighbors that that form triangles around that node, in order to answer 1.5-



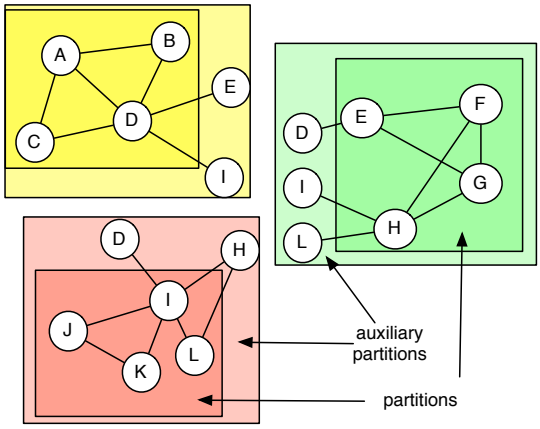
(a) Graph Snapshot.



(b) Random partitioning of graph snapshot with high number of edge-cuts.



(c) Min-cut partitioning of snapshot.



(d) Min-cut partitioning of snapshot with edge-cut replication and auxiliary storage strategy.

Figure 4.3: Graph partitioning using min-cut strategy along with 1-hop replication and the use of auxiliary micro-deltas improves 1-hop neighborhood retrieval performance without affecting the performance of snapshot or node retrieval.

degree ego [1] network queries effectively without replication of entire node. The choice of whether to employ these strategies can also be made based on the workload, and the frequency of neighborhood retrieval queries.

[1] A *1.5 degree ego network* of a node is an extension of the 1-hop ego network with mutual connections between the node's neighbors.

4.2.6 Dynamic Graph Partitioning

We mentioned previously that the horizontal partitioning of snapshots and eventlists, enable efficient retrieval of smaller portions of the graph, such as nodes or neighborhoods, without the need to fetch the entire snapshot. The key idea is to keep the size of each micro-delta (and each micro-eventlist) about the same and bounded by a number that dictates the latency for fetching a node or neighborhood. In a time-evolving graph, however, the size and topology of the graph change with time. So, a partitioning scheme suitable for a snapshot at a time point may not be suitable at a time later. Ideally, we could repartition or update a partitioning frequently so as to maintain a good partitioning[2] as per a particular notion of a graph partitioning heuristic. However, we also require a map for tracking partitions at each time we change the partitioning scheme. Maintaining and looking up that map as frequently as the changes in the graph is highly inefficient. Hence, we divide the history of the graph into time spans, where we keep the partitioning consistent within each time span, but perform it afresh it at the beginning of each new time span. This gives rise to two problems, described briefly as follows. Firstly, given a graph over time span, $T \in [t_s, t_e)$, find the graph partitioning that minimizes the edge cuts across all time points combined. Secondly, to determine the appropriate points for the end of a time span and the beginning of a new one, with respect to over all query performance. We discuss these problems below.

Static graph partitioning for an undirected and unweighted graph $G = (V, E)$ into

[2]Optimal graph partitioning is NP-complete. We use the adjective "good" in reference to the quality of a partitioning, with respect to the respective graph snapshot partitioning heuristic used in a particular algorithm.

k partitions is defined as follows. Each node $v_i \in V$ is assigned a partition set P_r such that $0 \leq r < k$. The constraint is that $\left\lfloor \frac{|V|}{k} \right\rfloor \leq |P_r| \leq \left\lceil \frac{|V|}{k} \right\rceil$, i.e., the partitions are more or less equal in size. The number of edge cuts across partitions are intended to be minimized, i.e., a count of all edges whose end points lie in different partitions. For a weighted graph, the edge cut cost is counted as the sum of the edge weights, which pushes stronger relationships (with higher edge weights) to be preferred for being in the same partition over the lesser weighted ones. Also, in case of a node weighted graph, the partition set count can be determined using the node weight. Different graph partitioning algorithms work under these constraints using one or the other heuristic, as described before.

For a dynamic graph partitioning, we consider an edge and node weighted, undirected time evolving graph, without the loss of generality. Consider the following: graph $G^T = (V^T, E^T, W_E^T, W_N^T)$ where, $T \in [t_s, t_e)$, is the time range for which we find a single partitioning; V^T, E^T, W_E^T, W_N^T are the set of vertices, edges, edge weights, node weights over time T, respectively. Our partitioning strategy involves projecting the graph over time range T to a single point in time using a *time collapsing* function Ω, there by reducing the graph G^T to a static graph, $G_T = \Omega(G^T)$. The constraint on function Ω is that G_T must contain all the vertices that existed in G^T at least once in G^T. Using G_T, we can employ static graph partitioning to find a suitable partitioning technique in the following manner.

The choice of Ω function determines how well the G_T is a representation of G^T. Let us consider a few different options. (1) Median: consider the time point t which is the median of the end points of T. The edges and weights in G_T are the edge weights in

G^t. (2) Union-Max: for an edge that existed at any time in G^T, we include it in G_T such that its weight is the maximum value from all time points in G^T. (3) Union-Mean: for an edge that existed at any time in G^T, we include it in G_T, where its weight is the weighted average (time fraction) of the edge weights in G^T. Non existence of an edge during a time period counts as a 0 contribution for the respective time period. (4). For any of the cases above, the node weight, w_n, can be defined independently of the edge set and edge weights. We consider three options as follows. (1) $w_n = 1$ for each nodes n in G_T; (2) $w_n = degree(n)$ for each node n in G_T; (3) $w_n = \overline{degree(n)}$, i.e., average degree over T.

Given these different heuristic combinations, we plan to study their empirical behavior and use the apparently most suitable one for TGI partitioning. The default TGI partitioning uses Union-Max for edge weights and uniform node weights.

We argue that this style of partitioning that involves first projecting a temporal graph to a static one, followed by conventional forms of static graph partitioning, is better than other conceivable alternatives. One such alternative way of doing it is to determine the partitioning at different time points in T, say P^T and then reducing P^T to P_T, a single partitioning scheme. This approach has the following major disadvantages. Firstly, the output partitions from a a static graph partitioning algorithm for two versions of graph G^T, say G^1 and G^2 are not aligned, even when the two snapshots are similar to a large extent. This is attributed to some degree of randomness associated with graph partitioning algorithms. This makes it infeasible to combine P^1 and P^2 in to a single result. Secondly, this approach is much more expensive compared to our approach, because it involves computing T orders of partitions. Another alternative approach is to use one of the *online*

graph partitioning algorithms, which updates a partition set for a graph upon a small change in the graph. However, the output of such an approach only gives us partitioning schemes at different time points. The partitions across time are better aligned to each other than the previous approach, but we would still need to compute a combined partitioning from all available partitions, and the notion of time collapsing is inevitable. Secondly, the partitioning results from incremental graph partitioning are often inferior compared to the batch mode of partitioning for obvious reasons.

Determining the appropriate number and the exact boundaries of time-spans is another important issue. The need for creating higher number of time-spans and hence reducing the duration of a time-span is to maintain healthier partitioning. Let the hit taken on query latencies (assuming a certain query load Q) due to a subpar snapshot partitioning be, $f(T)$. This hit is generally incurred on k-hop queries, without replication, due to higher number of micro-delta seeks. In case of replication across partitions, the degree of replication increases with inferior partitioning, and leads to indirect impact on query latencies. On the other hand, there is need to create longer time-spans because the version queries require multiple micro-deltas, at different time points. Higher the changing number of partitions over query's time interval, say t, higher the query latency. Let us say that for an average query time interval (again, as per a specific query load), the gain due to longer time spans is, $g(T)$. The appropriate length of a average time-span hence is the solution of the maxima of $g(T) - f(T)$. In practice, uniform time-span length in numbers of the number of events is perhaps the most convenient. While the models of f and g are complex, a good number for size of T can be observed empirically.

4.2.7 Fetching Graph Primitives

Now, we briefly describe how the different types of retrieval queries are executed.

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 their micro-partitions) along that path 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.

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 micro-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 it's

neighbors, fetch them, and so on. 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. The first one is to request 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. The second one is the request for the state of the neighborhood at multiple specific time points. This translates to the retrieval of multiple single neighborhoods fetch tasks.

4.3 Evaluation

The efficacy of a system such as HGS is determined by two factors. First, whether the functionality provided by the system can be easily used in achieving a set of meaningful objectives, in this case, a wide range of analytical tasks on historical graphs. Second, whether the system performs these objectives efficiently in terms of time and other resources. In this section, we evaluate TGI on the second factor.

To recap, TGI is a persistent store for entire histories of large graphs, that enables fast retrieval for snapshots, subgraphs, versions of nodes or neighborhoods, etc. The range of different retrieval primitives pose two distinct trade-offs in indexing and retrieval. First,

the size of a subgraph can vary from a node to an entire graph. Second, the history of a subgraph to be retrieved can range from one time point to the graph's entire time span. We show how TGI captures these trade-offs. Note that we do not provide experimental results on the internals of snapshot retrieval which have been thoroughly explored in the context of DeltaGraphs (Section 3.7). Additionally, we report results for demonstrating its scalability over large data and heavily parallel fetching.

Datasets and Notation: We use four datasets: (1) Dataset 1 is the Wikipedia citation network consisting of 266,769,613 edge addition 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 600 million events; (3) Dataset 3 is obtained similarly from Dataset 1 by adding 733 million events, making the total around 1 billion events; (4) Dataset 4 is derived from a snapshot of the Friendster gaming network by adding synthetic dates at uniform intervals to 500 million events with a total of approximately 37.5 million nodes and 500 million edges.

Some of the key parameters that are varied in the experiments include: 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 size of the Spark analytics cluster (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.

Snapshot retrieval: Figure 4.4 shows the snapshot retrieval times for Dataset 1 for different values of the parallel fetch factor, c. As we can see, 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 larger number of machines), the linear speedup would be seen for larger values of c (this is also corroborated by the next set of experiments). The snapshot retrieval times for dataset 4 can be seen in Figure 4.5.

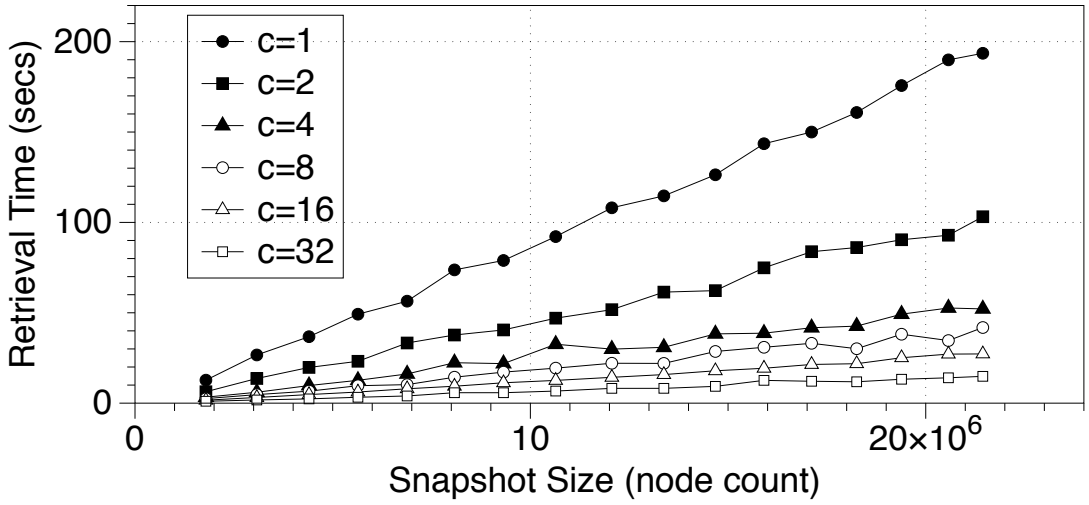


Figure 4.4: Snapshot retrieval times for varying parallel fetch factor (c), on Dataset 1; $m = 4$; $r = 1, ps = 500$.

Figure 4.6 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

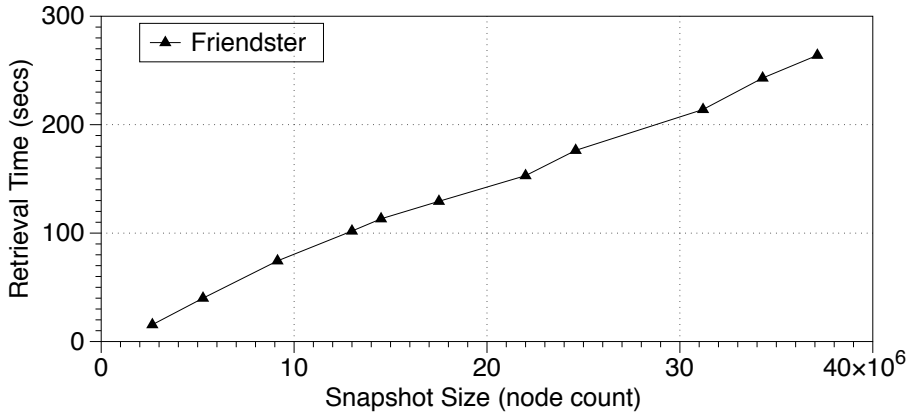


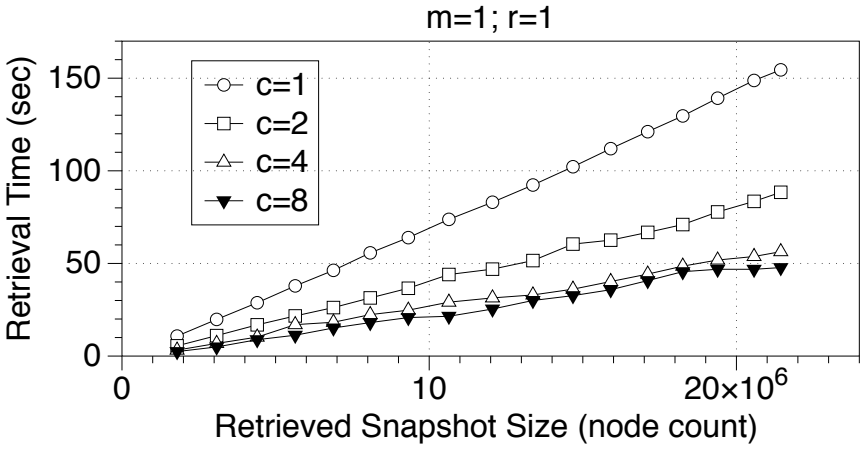
Figure 4.5: Snapshot retrieval times for Dataset 4; $m = 6$; $r = 1, c = 1, ps = 500$.

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.

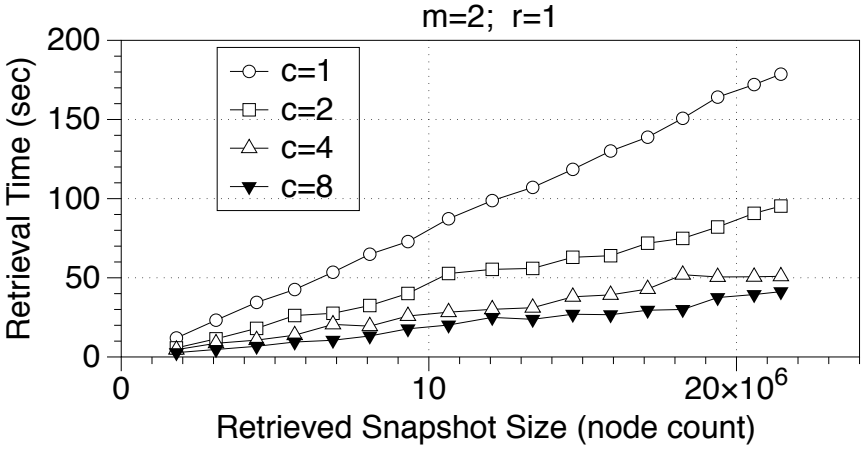
Further on snapshot retrieval, Figure 4.7a shows that for large snapshots, bulk of the time is spent in post-processing, mostly *deserialization* of the deltas, compared to fetching the binary encoded deltas from the data store. The net effect of Cassandra compression for deltas is negligible for TGI. Figure 4.7b is representative of the general behavior.

Size of the delta partitions (or the number) affects the performance the snapshot retrieval performance only to a small degree as seen in Figure 4.7c. This occurs due to a the TGI design which makes sure that all the partitions of a delta (micro-deltas) are stored contiguously in a cluster. This demonstrates that TGI is a superset of DeltaGraph where we are able to handle other queries along with efficient snapshot retrieval.

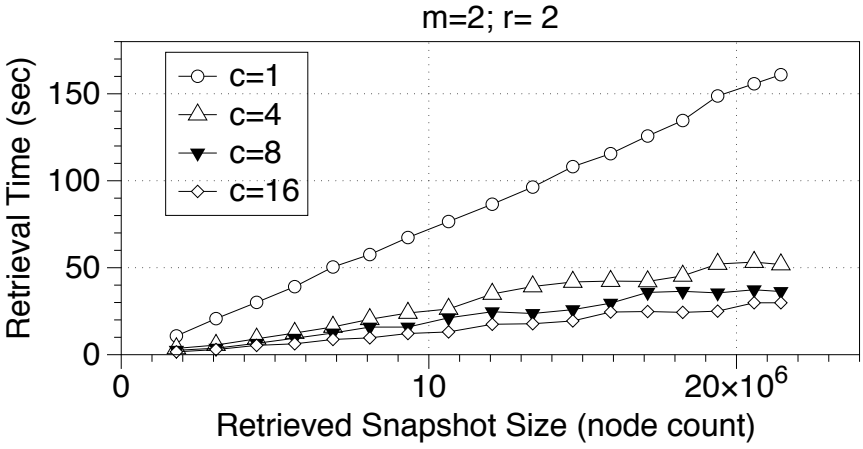
Node History Retrieval: Smaller eventlists or partition sizes provide a lower latency



(a) m=1; r=1; ps=500

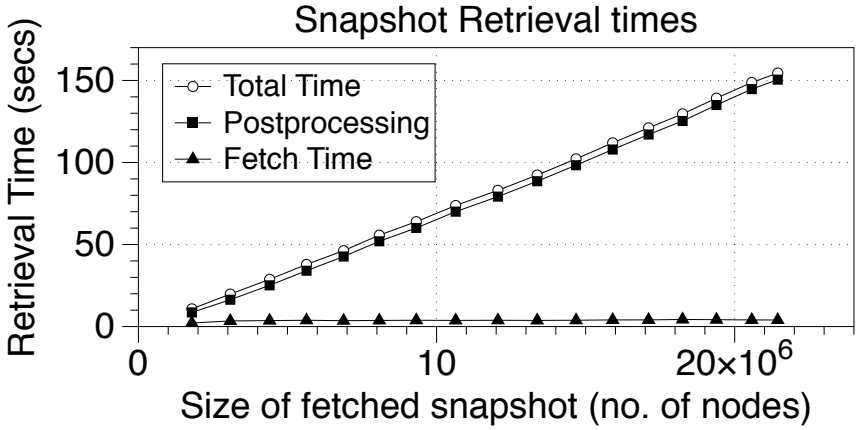


(b) m=2; r=1; ps=500

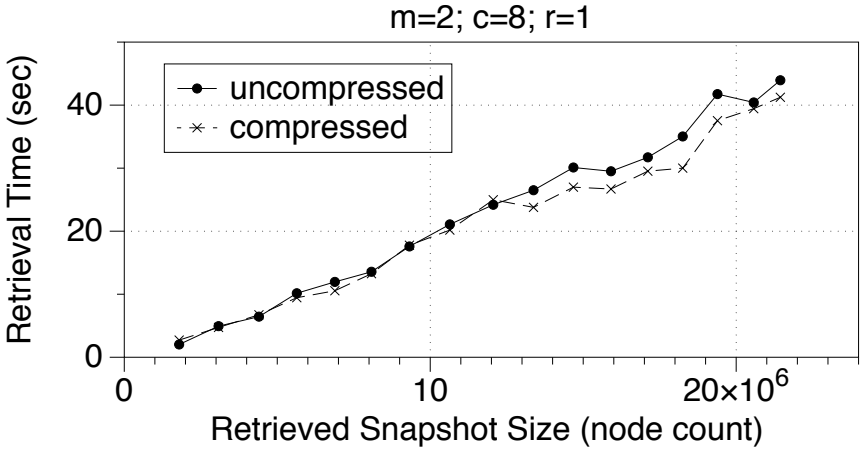


(c) m=2; r=2; ps=500

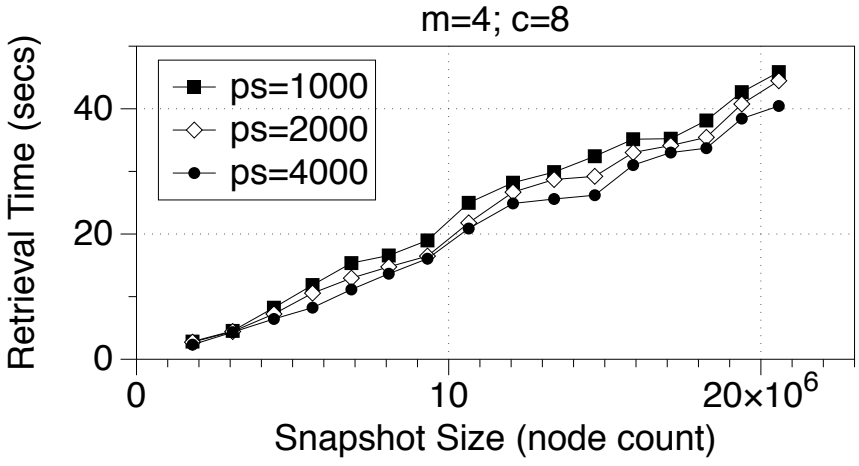
Figure 4.6: Snapshot retrieval times across different m and r values on Dataset 1.



(a) Fetching and postprocessing times.



(b) Compressed vs. uncompressed delta storage.



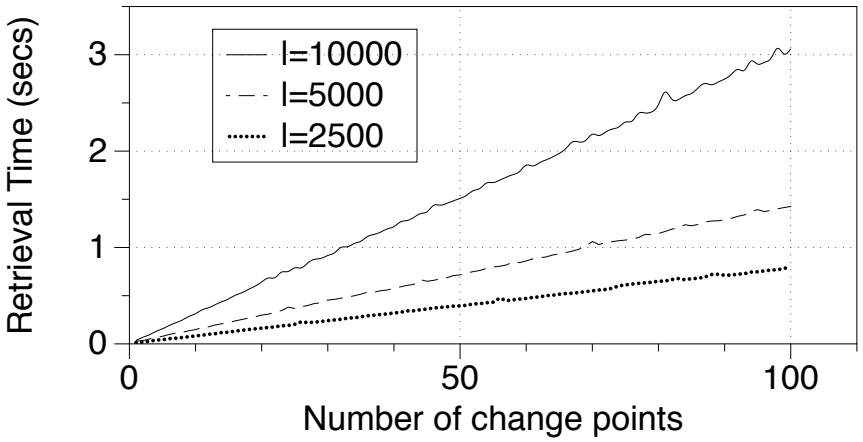
(c) Effect of partition sizes.

Figure 4.7: Snapshot retrieval across various parameters.

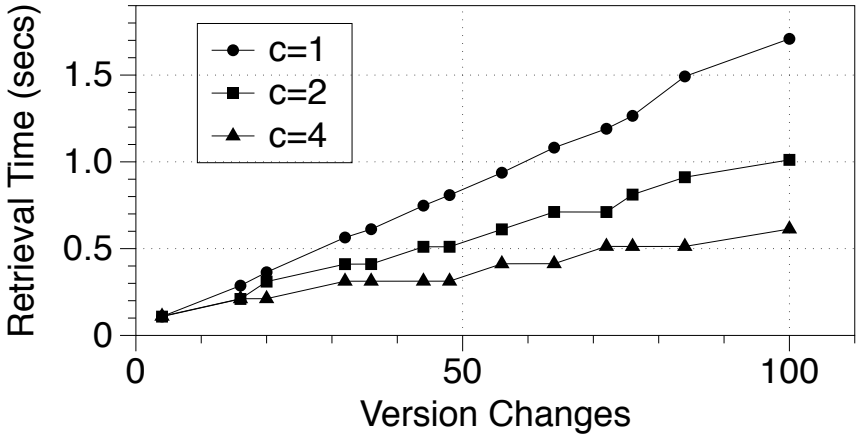
time for retrieving different versions of a node, which can be seen in Figure 4.8a and Figure 4.8c, respectively for Dataset 1. This is primarily due to the reduction in work required for fetching and deserialization. A higher parallel fetch factor is effective in reducing the latency for version retrieval (Figure 4.8b). Note that the impact of varying partition sizes on version retrieval performance is opposite to the impact on snapshot retrieval performance; hence, we are presented with a trade-off in selecting the appropriate partition size. However, smaller eventlist sizes benefit both version retrieval and snapshots. Node version retrieval for Dataset 4 shows a similar behavior, which can be seen in Figure 4.9.

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 4.10a for 1-hop neighborhood retrieval latencies. As discussed in Section 4.2, 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, which can be reasoned based upon similar speed-ups for 1-hop neighborhoods.

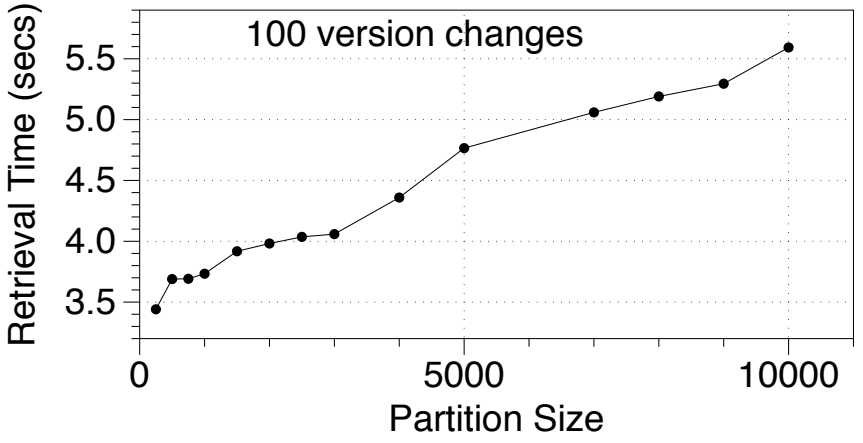
Increasing Data Over Time: We observed the fetch performance of TGI with an increasing size of the index. We measured the latencies for retrieving certain snapshots upon varying the time duration of the graph dataset, as shown in Figure 4.10b. Datasets 2 and 3



(a) Effect of eventlist size, l.



(b) Speedups due to parallel fetch factor, c.



(c) Effect of partition sizes.

Figure 4.8: Node version retrieval across various parameters for Dataset 1.

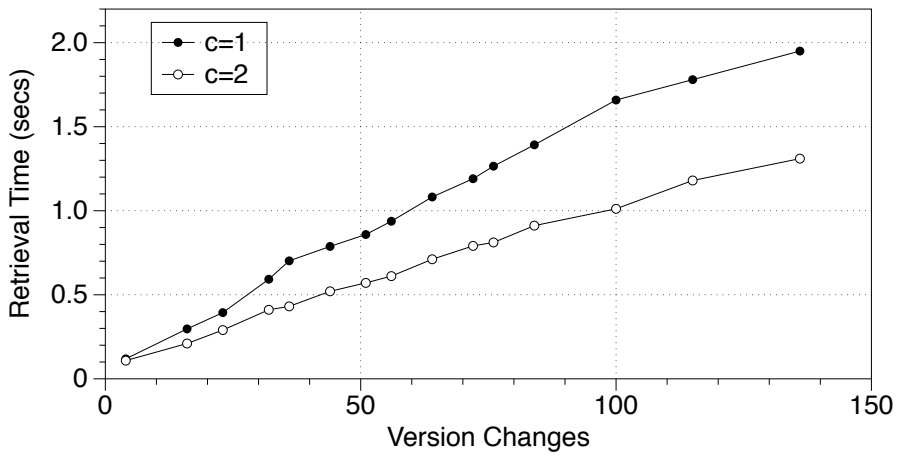
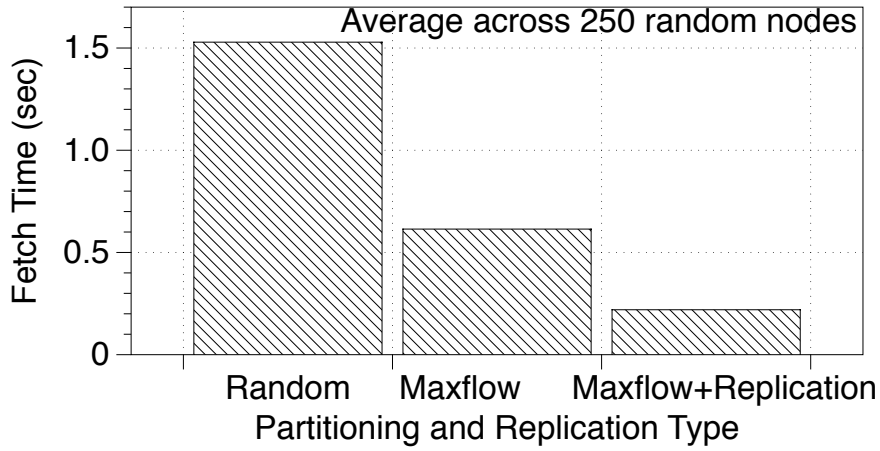


Figure 4.9: Node version retrieval for Dataset 4; $m = 6$; $r = 1, c = 1, ps = 500$.

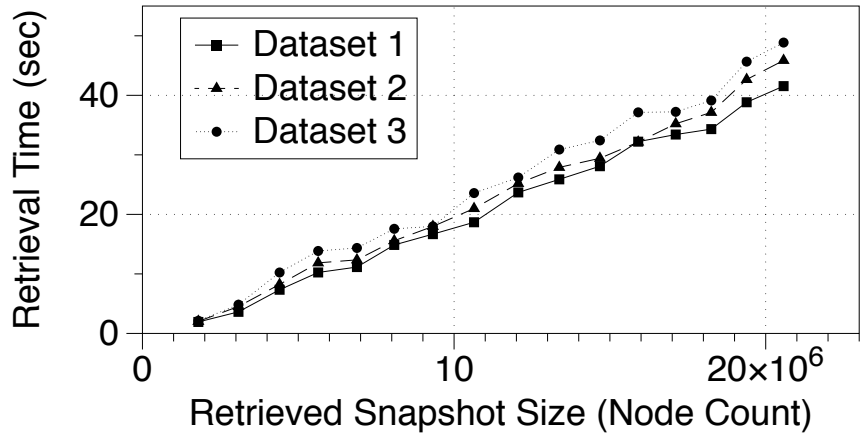
contain additional 333 million and 733 million events over dataset 1, respectively. Only a marginal difference in snapshot retrieval performance demonstrates the scalability of our system for large datasets.

4.4 Conclusion

Most real-world networks are large and highly dynamic. This leads to creation of very large histories, making it challenging to store, query, or analyze them. In this chapter, we presented a novel structure, *Temporal Graph Index (TGI)*, 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. TGI is designed for elastic storage and query processing, suitable for most real world graphs. TGI builds on top DeltaGraph which is optimized to handle snapshot retrieval. TGI, on the other hand provides efficient access to not only snapshots, but version queries as well, and is comparatively efficient in fetching graph components of varying sizes, from a node to a an entire snapshot. Certain aspects of TGI design, such as different forms of



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



(b) Snapshot retrieval for varying sizes of datasets.

Figure 4.10: Experiments on partitioning type and replication; growing data size.

partitioning in dynamic graphs enable it to conquer topological and temporal skew, typical of evolving graph datasets, and makes it suitable for balanced distributed computing and elastic scalability.

We briefly summarize a couple of the implicit and explicit assumptions made by the Temporal Graph Index. First, it is assumed that the input temporal graph data is present entirely in the form of time annotated events. This means that missing time-points or missing events are not acceptable as a part of a valid input dataset. This also

implies that a dataset provided in the form of timestamped snapshots is best interpretable as a set of several events occurring at the time of the corresponding snapshot, derived by a computing a delta from the previous or empty snapshot. Second, it only supports "appends" and not modifications, i.e., changes to a later state than the latest one but not the ones corresponding to time points already covered by the index.

Chapter 5: Temporal Graph Analytics

5.1 Introduction

Historical graphs present a rich set of analytical opportunities. Distinct notions of static graph analysis and temporal data analysis can be applied individually, or in novel combinations, to derive various insights from time-evolving graphs. However, current batch analytics frameworks like Map-Reduce or Spark lack built-in support for temporal reasoning, and thus an analyst is burdened with figuring out how to express a temporal analysis task using the primitives supported by those frameworks, leading to inefficiencies and duplication of functionality. Our objective is to provide a principled framework for conveniently expressing such tasks over large volumes of data. In this chapter, we look at some systematic aspects concerning temporal graph analytics.

Consider the example of snapshot based evolution analysis of a graph, such as pagerank over 100s of snapshots. Using DeltaGraph or TGI, we are able to efficiently fetch multiple snapshots from a disk resident index into memory. However, storing 100s of (potentially) very large graphs requires great amounts of primary memory. Selectively fetching portions of a graph into memory on demand is not a solution as most graph algorithms do not benefit from locality caching due to the inherent structure of the graph. We could process one snapshot at a time but the temptation to carry out multiple analytical

tasks simultaneously, generalizes the given problem of memory bottleneck. One of the questions we address in this chapter is, "does the overlap between different versions of the graph snapshots, offer an opportunity for compressed storage of the snapshots, without compromising direct access to process the graphs?"

In order to scale temporal graph analytics to large volumes, it is imperative that we move towards the cluster computing paradigm. Due to the nature of graph analysis algorithms, a Map-Reduce style framework such as Hadoop is not suitable for iterative computation. Apache Spark [142] is a popular in-memory cluster computing framework which supports functional programming style library. In this chapter, we explore the constructs of a system for performing temporal graph analysis tasks on a framework built over Spark.

5.1.1 Contributions

In this chapter, we present two novel components:

- **GraphPool** 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 interior or leaf nodes in the DeltaGraph, but may not correspond to any valid graph snapshot GraphPool exploits redundancy amongst the different graph snapshots that need to be retrieved, and considerably

reduces the memory requirements for historical queries.

- **Temporal graph Analysis Framework (TAF)**: We provide an expressive library to specify a wide range of temporal graph analysis tasks and to execute them at scale in a cluster environment. The library is based on a novel set of *temporal graph operators* that enable a user to analyze the history of a graph in a variety of manners. The execution engine itself is based on Apache Spark, a large-scale in-memory cluster computing framework.

Outline: We continue the rest of this chapter as follows. In section 5.2, we provide a background of relevant work for temporal graph analytics systems. In Section 5.3, we present the details of GraphPool including the design, update, application using visual analytics, and finally experiments supporting the efficacy of GraphPool. In Section 5.4, we present the Temporal Graph Analysis Framework, listing the operators and operands provided in the library, followed by the details of the system design and experiments for its performance. We conclude the chapter with a summary and a relevant discussion in Section 5.5.

5.2 Background

In this section, we discuss various network analytics routines, parallel graph processing systems, and temporal data analytics systems. We talk about the opportunities provided by them along with their limitations in order to identify the appropriate constructs of a temporal graph analytical system.

Over the last decade, several network analysis packages have surfaced. Most of

these packages implement popular graph theoretic algorithms such as shortest paths, centrality determination, clustering, decomposition, flows, etc., that can be programmatically employed to analyze a graph, combined with analytical tools such as graph visualization and curve plotting. JUNG (Java Universal Network/Graph Framework) [99] is a Java based library which provides visually interactive graph displays and various graph metric computations. SNAP [86] is a C++ based package (with a Python interface) with similar graph mining and analysis features, known for scaling well on large graphs. Blueprints [4] is a graph analysis middleware that provides a common interface for several storage backends (such as Neo4J [10], OrientDB [130], Titan [15], MongoDBGraph [46] and others) with graph analysis packages (such as JUNG, SAIL [12], Furnace [7] and others) or languages (such as Gremlin [8]). Most of these routines, however, are not designed to scale up for a cluster environment. Also, none of them provide support for temporal analytics.

Graph processing frameworks such as Pregel [90] (and its open source version, Giraph [1]), GraphLab [88] and recently, GraphX [56], are designed to execute tasks on large graphs. These frameworks are based on a node-centric functional programming paradigm and the underlying graph processing in the cluster uses specific synchronization mechanisms to communicate across nodes for tasks such as traversal or simply accessing a neighbor's state. While these frameworks are able to scale up to large graphs, they are limited by the absence of rich libraries that can help compute a wide range of algorithms. A part of reason for this limitation lies in the their design that does not generalize well to parallelizing a wide variety of graph algorithms. None of these frameworks support temporal analytics.

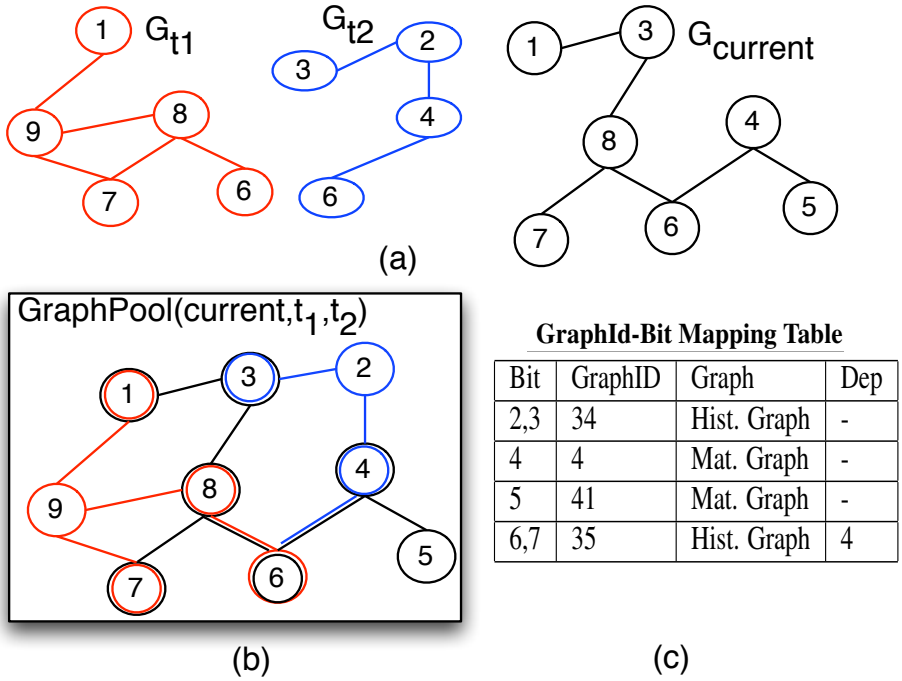
Temporal graph processing also involves the handling of streaming graph data.

However, the nature of the queries in that domain is quite different from the general temporal queries such as evolution, comparison or historical ones. In stream-data processing, the focus is on sliding window queries, i.e., evaluating queries such as aggregates on a fixed length window of time whose end point is usually the current or the latest time. When the updates "stream in", the query result is continuously updated with the updated state of the window. Such analysis is applied in case of continuous monitoring systems and there usually exists a "single pass" constraint, i.e., the stream is read once and never stored; the query processing in this domain is fairly specific and solutions applied are mostly based on approximations. A comprehensive account of graph streaming work is out of the scope of this chapter, so we refer the reader to a few interesting pointers, such as PageRank estimation [114], outlier detection [20], sketches [23], and a recent survey of algorithms on graph streams [93]. The premise of our analytical objectives is quite different, that is, we store the entire history and strive to extend all static network analytics to a temporal dimension.

In order to design a framework for large scale temporal graph analysis, we borrowed concepts and artifacts from many of the different systems alluded to in the discussion above.

5.3 Memory Efficient Loading of Multiple Versions

In this section, we describe the key ideas behind *GraphPool*, which stores multiple graphs in-memory in an overlapping manner, yet providing direct access each individual graph.



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

Figure 5.1: (a) Graphs at times $t_{current}$, t_1 and t_2; (b) GraphPool consisting of overlaid graphs; (c) GraphID-Bit Mapping Table

5.3.1 GraphPool

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 5.1). 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 5.1(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 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 5.1(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 bitmap 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.

5.3.2 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.

5.3.3 Visual Analytics on Historical Graph Snapshots

We describe *HiNGE (Historical Network/Graph Explorer)* [74], an analysis and visualization tool for historical networks. HiNGE has been implemented in Java; it uses DeltaGraph and GraphPool libraries for storage and graph representation respectively and the JUNG[1] [99] library for network visualization. While the basic DeltaGraph provides temporal indexing and snapshot retrieval over the evolving network, its auxiliary indexing capabilities support features such as Node Search and Pattern Finding. Figure 5.2 de-

[1] `http://http://jung.sourceforge.net.`

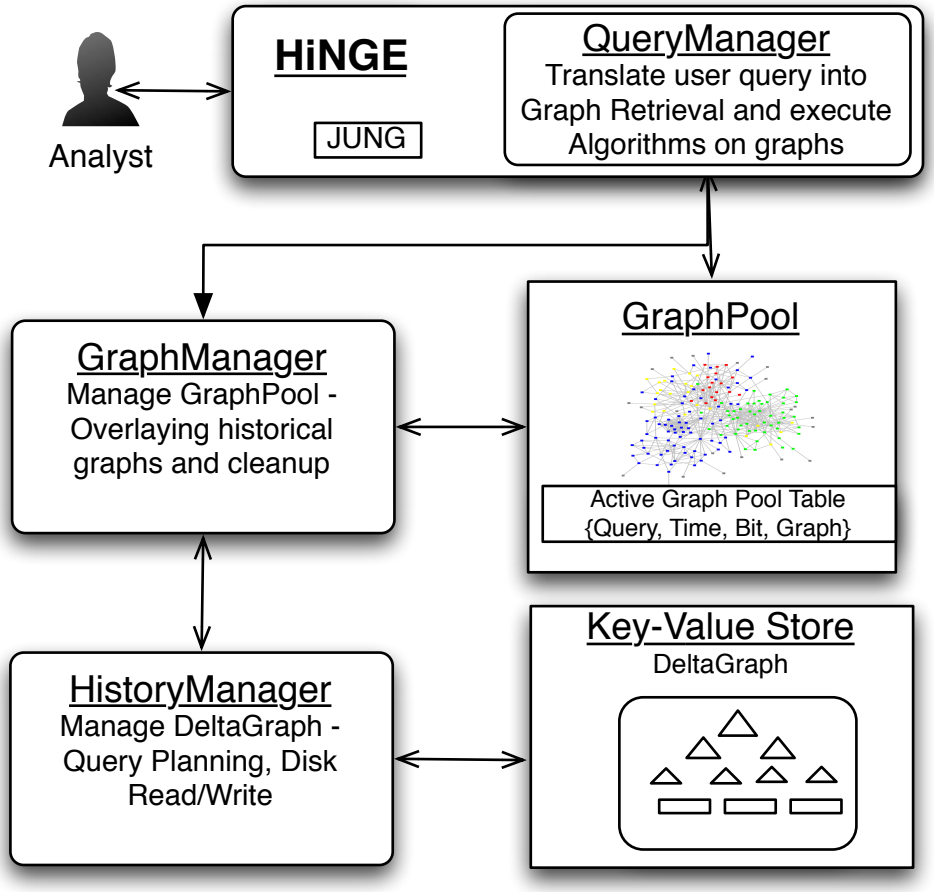


Figure 5.2: System Architecture: HiNGE, DeltaGraph and GraphPool.

scribes the interaction between various components of the overall system. Further in this section, we describe the features and usability of HiNGE.

5.3.3.1 Features

Features supported by HiNGE may be classified as exploration tasks, evolutionary queries and search queries.

- **Exploration:** As the name suggests, is built for an starter's insight into the network. It is meant to provide the user with a basic level of control to "browse" through the

network while observing the structure and properties of the graph, and perhaps, certain anomalies as well. It is considered to be the stepping stone for more specific enquires into the nature of the network. Exploration of a temporal graph is enabled using – (a) time-slider, (b) an interactive, zoomable snapshot viewer, and (c) metric calculator. The *time-slider* is an an interactive timeline, that the user can adjust to go to a specific time of interest. The *snapshot viewer* presents a view of the graph at the desired time as indicated by the time-slider. The user may pan, zoom or rotate the pane with mouse operations to focus on the area of interest in the graph. The layout, color and other factors of appearance of the graph can also be changed by customizing the choices in the *Settings* menu. The *metric calculator* provides the choice of several metrics such as PageRank, Betweenness Centrality, Clustering Coefficient for the vertices of the network at the time indicated by the time slider. The metric values may be chosen as a part of vertex labels in the snapshot view, or used to make the graph display more appropriate, and the *top/bottom-k* valued vertices are displayed on the side. These can be seen in Figure 5.3.

- **Query:** The Query mode is meant to provide a comparative temporal analysis of the vertices of interest. During the exploration phase, a user may find certain vertices of interest. The query interface allows for a detailed evolutionary analysis of those vertices in particular. It shows the structural evolution as well as the change in the scores of interest, such as the *clustering coefficient*. The following are the components used in the query mode – (a) query, (b) node evolution, (c) metric evolution. The *query* specifies the vertex, the start and end times, the metric of interest

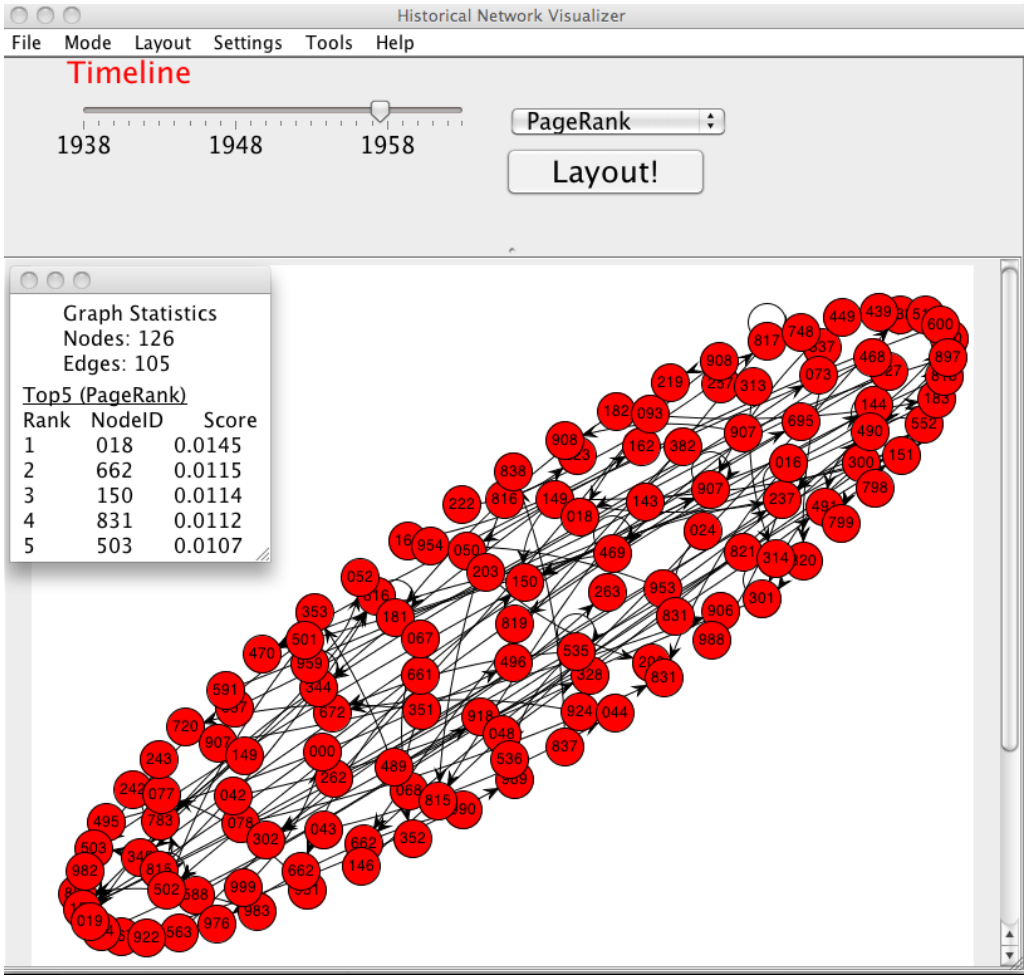


Figure 5.3: Temporal Exploration using *time-slider*

and number of time points used in the comparison. The *node evolution* is shown as the evolution of the node and its neighborhood at different points in time. The *metric evolution* plots the metric value over time. These can be seen in Figure 5.4.

- **Search:** An interesting and slightly different kind of query is *Subgraph Pattern* finding. The user may specify the query by drawing the structure of a subgraph, assigning labels to the nodes and specifying the time interval during which to perform the search. The result lists all the matches found for the query, i.e., the subgraph layouts and times at which the particular subgraph exists. This is implemented in the

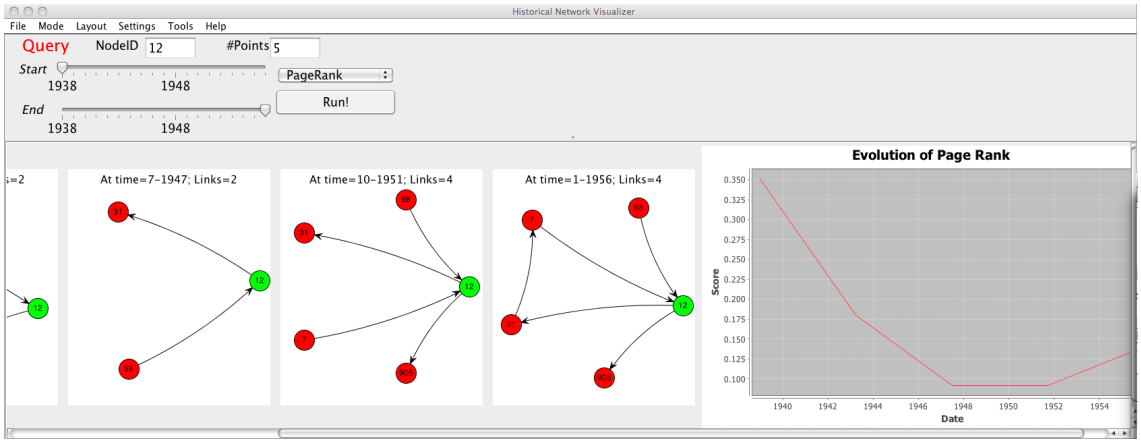


Figure 5.4: Temporal Evolution of *node 12* in green over three points in time: Evolution of node's neighborhood and PageRank.

DeltaGraph though an auxiliary path index. A very useful feature is *Node Search*, which helps the user find nodes based upon attribute values. This is implemented using the an auxiliary inverted index in DeltaGraph. Hence, the user may constrain the search by specifying a time interval. Figure 5.5 shows the node search and sub-graph pattern search features. By keeping the time range open, we can specify a search across all times; whereas, one with the end point and start point as the same refer to a search query in that particular snapshot. More generally, it could be a time period.

Working with HiNGE: The expected input graph specification is as described in [73]. The evolving network is described as a set of chronological events. Each node is required to have a unique identification, the *nodeid*. Nodes and edges may carry any number of attributes, for example, name, weight etc. While specifying the node in the query, we are required to mention the*nodeid*. Node search should be used to locate a *nodeid*, when only the attributes of the node are known. Here is a list of the major options/parameters, all of

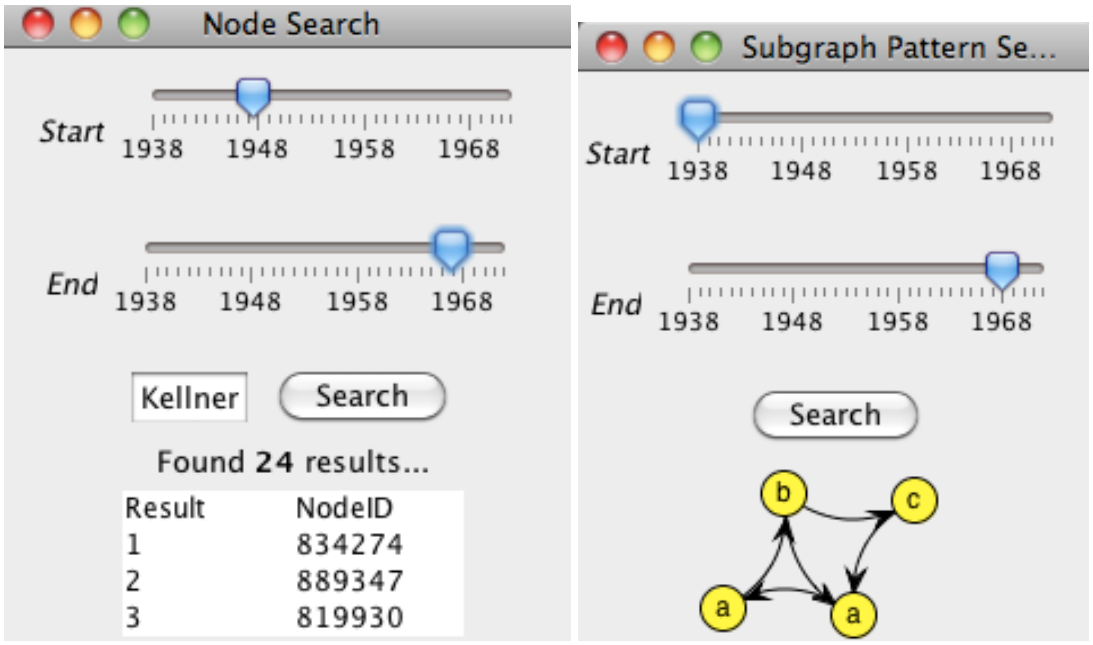


Figure 5.5: (a) Node Search (b) Subgraph Pattern Search

which can be accessed from one of the menus, such as follows:

1. A new network can be loaded from *File* > *Load Network*.

2. The graph appearance may be altered by changing the default layout (*Layout* > *Algorithm*), colors (*Layout* > *Color*) and the shape and size (*Layout* > *Shape and Size*). The latter may be based on the metric scores of the nodes.

3. Switch from the Explore mode to the Query mode by *Mode* > *Query* and vice-versa.

4. Set the k and bottom or top in *top/bottom k* by going to (*Settings* > *Top/Bottom k*).

5. Perform a node search by going to *Tools* > *Search*. Perform a subgraph pattern query by going to *Tools* > *Find Patterns*.

6. Find the documentation at *Help*.

5.3.4 Experiments

GraphPool memory consumption:

Figure 5.6 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.

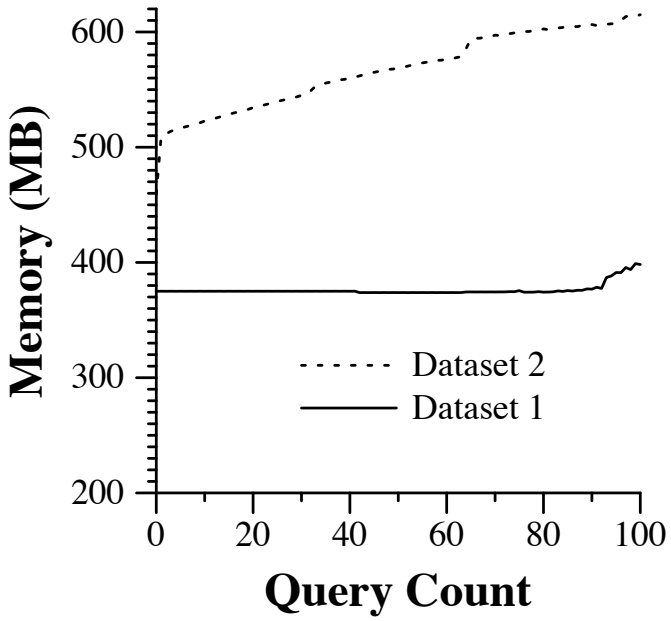


Figure 5.6: Cumulative GraphPool memory consumption.

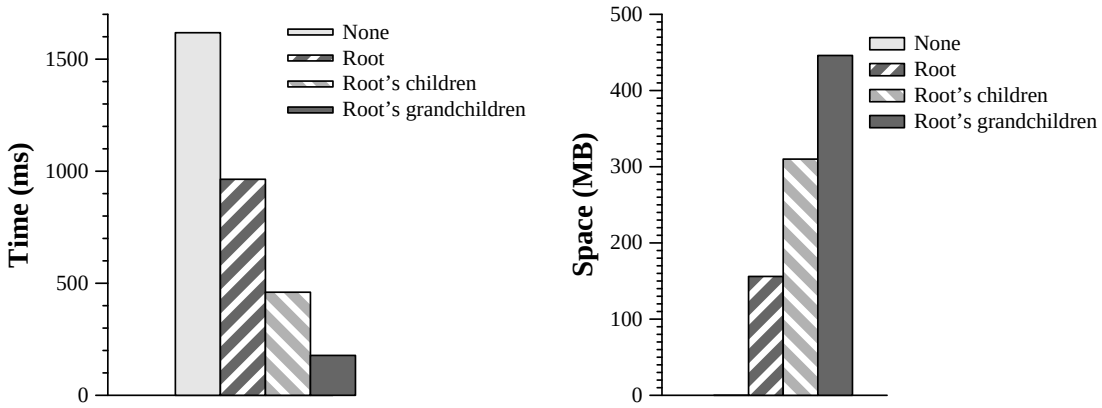
For Dataset 2, if the 100 graphs were to be stored disjointly, the total memory consumed would be 50GB, whereas the GraphPool only requires about 600MB. The plot of Dataset 1 is almost a constant because, for this dataset, any historical snapshot is a

subset of the current graph.

The minor increase toward the end is due to the increase in the bitmap size, required to accommodate new queries.

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

Materialization: Figure 5.7 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 of higher memory consumption.



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

Figure 5.7: Effect of materialization

5.4 Temporal Analytics Framework (TAF)

In this section, we describe our proposed analytics framework, *Temporal graph Analysis Framework (TAF)*, that enables programmers to express complex analytical tasks on time-evolving graphs. We expose the operators in the framework though Python and Java APIs. TAF is built on top of Apache Spark, which enables parallel, in-memory execution.

5.4.1 Temporal Graph Analysis Library

In this section, we describe the set of operators for analyzing large historical graphs. At the heart of this library is a data model where we view the historical graph as a *set of nodes or subgraphs evolving over time*. The choice of temporal nodes as a primitive helps us describe a wide range of fetch and compute operations in an intuitive manner. More importantly, it provides us an abstraction to parallelize computation. The *temporal nodes* and the *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.

Operands: We begin with defining the two main data types that most operators operate upon.

Definition 8 (Temporal Node)**.** A *temporal node* (NodeT), $\mathcal{N}^T$, is defined as a sequence of all and only the states of a node $\mathcal{N}$ 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 9 (Set of Temporal Nodes)**.** A *SoN*, is defined as a set of r temporal nodes $\{\mathcal{N}_1^T, N_2^T \ldots N_r^T\}$ over a time range, $T = [t_s, t_e)$.

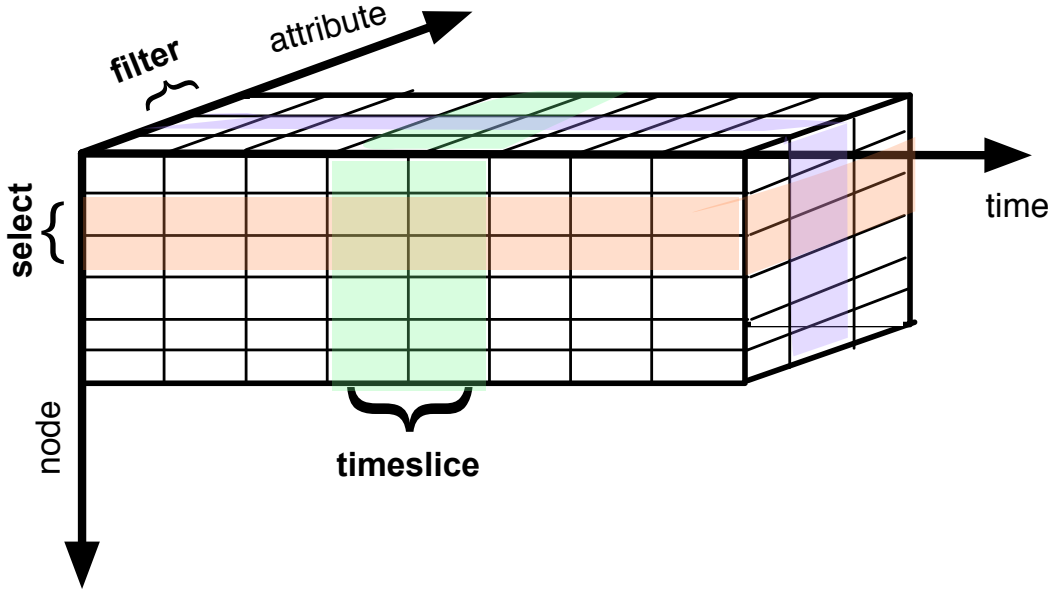


Figure 5.8: SoN: A set of nodes can be abstracted as a 3 dimensional array with temporal, node and attribute dimensions.

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

Operators: Below we discuss a subset of the temporal graph algebra operators supported by our system.

1. Selection: This operator accepts an SoN or an SoTS along with a Boolean function on the nodes or the subgraphs, and returns an SoN or SoTS as output. Selection performs *entity centric filtering* on the operand, and does not change temporal or attribute dimensions of the data.

2. Timeslicing: This operator 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 (we use SoN/SoTS to also represent sets of static nodes or subgraphs as a special case). The operator can also take a list of timepoints as input, returning a list of SoN or SoTS.

3. Graph: This operator takes in an SoN and returns an in-memory Graph object over the nodes in the SoN (containing only the edges whose both endpoints are in the SoN). An optional parameter, tp, may be specified to get a GraphS valid at time point tp. Note that the return Graph object is not distributed, and hence the user must make sure that it can fit in the memory of a single machine.

4. NodeCompute: This operator 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 (NodeComputeDelta) that operates on the delta between two versions of a node (subgraph), which the system can use to compute the output more efficiently. 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.

6. NodeComputeDelta: This 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) like `NodeComputeTemporal()`, (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, the method of computation in this method is different because it updates the computed 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: This 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: This 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: This abstractly represents a collection of temporal aggregation operators such as `Peak`, `Saturate`, `Max`, `Min`, and `Mean` over a scalar timeseries. The aggregation operations are used over the results of temporal evaluation of a given quantity over 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.

5.4.2 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.

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 `RDD<NoteT>` and `RDD<SubgraphT>`). For constructing the in-memory graph objects, we use the open-source TinkerGraph implementation [14], but other in-memory graph implementations could be used as well. TinkerGraph supports the Blueprints Graph API, and thus the graph objects that are created can be manipulated or analyzed using the Gremlin query language and other software packages built on top of Blueprints. 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 5.9 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

```
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 5.9: Examples of analytics using the TAF Python API.

the cost to create the index at query time is likely to offset any access latency benefits due to the index. An intuitive guess based upon examination of certain temporal analysis tasks is that its access pattern is most likely going to be in a 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 the details of these methods as their functionality is apparent from the nomenclature.

NodeComputeDelta: As defined in Section 5.4.1, `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 finding the fraction of nodes with a specific attribute value in a given `SubgraphT`. If this were to be 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 the incremental computation, each new version can be processed in constant time after the first snapshot, which adds up to, $\mathcal{O}(N+T)$. While performing the 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 changing 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 in Figure 5.10 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 the one shown in Figure 5.11(a). 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

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

labelCount = sots.NodeComputeDelta(fCountLabel)


...

def fCountLabel(g):
   labCount = 0
   for node in g.GetNodes():
      if node.GetPropValue('EntityType') == 'Author':
          labCount += 1
   return labCount
```

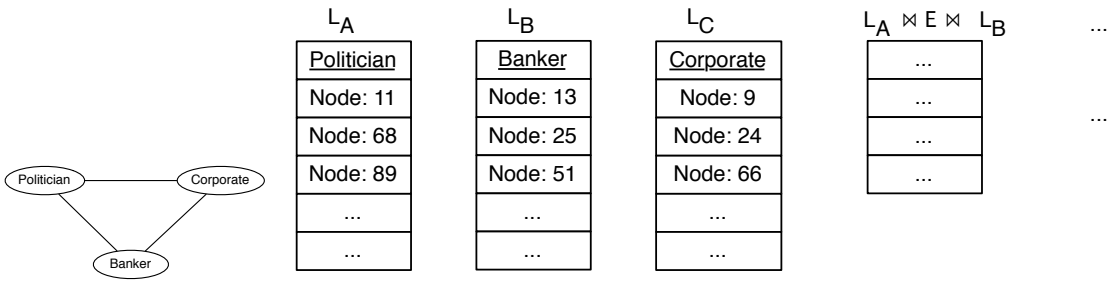
(a) Using `NodeComputeTemporal`

```
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, fCountLabelDel)


...

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



def fCountLabel(g):
   labCount = 0
   for node in g.GetNodes():
      if node.GetPropValue('EntityType') == 'Author':
          labCount += 1
   return labCount
```

(b) Using `NodeComputeDelta`

Figure 5.10: Incremental computation using different methods. Both examples compute counts of nodes with a specific label in subgraphs over time.

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



(a) Patten to be matched (b) Ongoing indexes need to be maintained to verify the pattern

Figure 5.11: Pattern matching in temporal subgraphs.

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 for using a considerable amount of algorithmic work available in literature on online and streaming graph query evaluation, respectively, to be applied to historical graph analysis. For instance, there is work on pattern matching in streaming [135, 52] and incremental computing [51, 133] contexts, respectively.

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. Couple of such examples can be seen in Figure 5.12.

Data Fetch: In a temporal graph analysis task, we first need to instantiate a *TGI connec-*

```
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
```

(a) Specifying the start, end and middle point of `SON` for an `Evolution` query.

```
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(),
        \ selectTimepointsAll)
print('Average membership in 2003,')
print(A=%s\tB=%s'%(mean(compAB[0]), mean(compAB[1])))

...


def selectTimepointsAll(sonA, sonB):
    time_arr = []
    ptsA = sonA.GetAllChangePoints()
    ptsB = sonB.GetAllChangePoints()
    time_arr = ptsA + ptsB
    return time_arr
```

(b) Specifying all the change points in two `SON`'s for a `Compare` query.

Figure 5.12: Using the optional timepoint specification function with evolution and comparison queries.

tion 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 5.4.1, 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.

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 5.13. The protocol is briefly described in the following ordered steps:

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 the fetch instructions and the former is made aware of the active TGI query processor nodes.

3. Parallel fetch starts at the TGI cluster.

4. The TAF master instantiates a TGIDriver instance at each 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 the end of fetch at TGI, the individual TGI nodes transfer a portion of the SoN to an RDD on the corresponding TAF nodes.

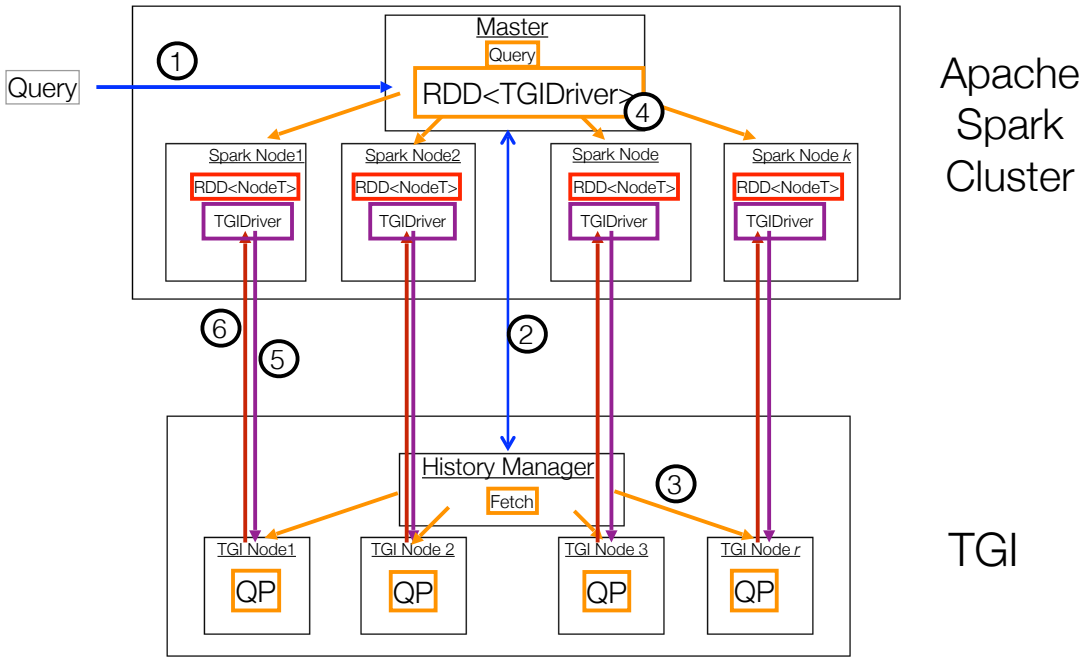


Figure 5.13: A pictorial representation of the parallel fetch operation between the TGI cluster and the analytics framework cluster. The numbers in circles indicate the relative order and the arrowheads indicate the direction of information or data flow.

5.4.3 Experiments

Dataset: We use the Wikipedia citation network consisting of 266,769,613 edge addition events from Jan 2001 to Sept 2010. At its largest point, the graph consists of 21,443,529 nodes and 122,075,026 edges.

Conducting Scalable Analytics: We examined TAF's performance through an analytical task for determining the highest local clustering coefficient in historical graph snapshot. Figure 5.14 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.

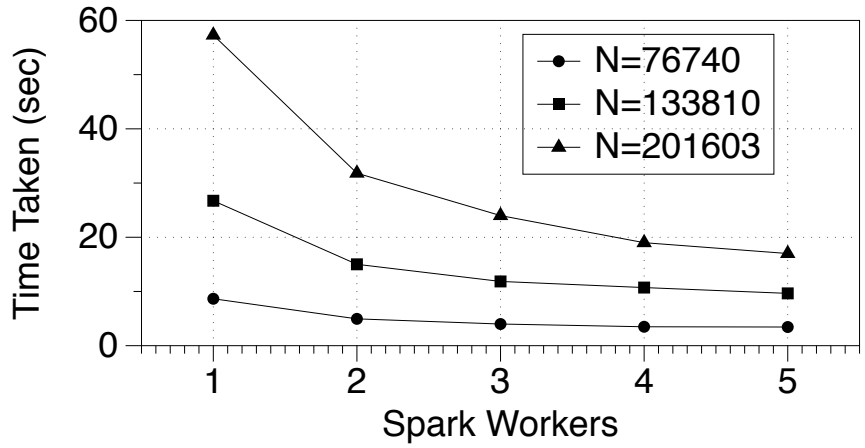


Figure 5.14: TAF computation times for Local Clustering Coefficient on varying graph sizes (N=node count) using different cluster sizes.

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

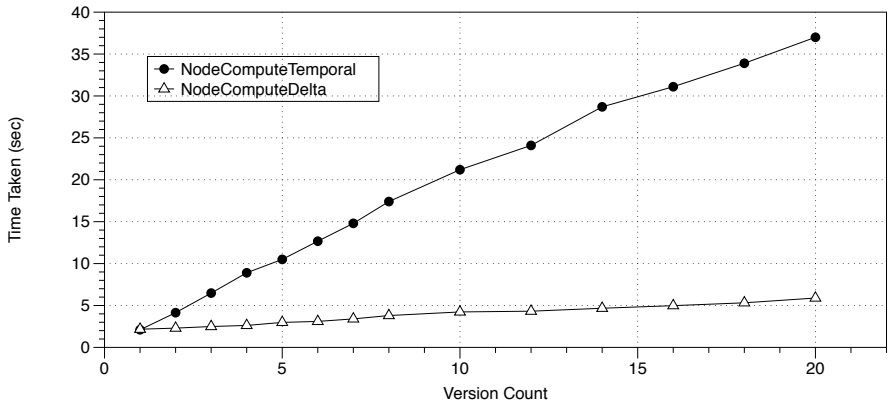


Figure 5.15: Label counting in several 2-hop neighborhoods through version (`NodeComputeTemporal`) and incremental (`NodeComputeDelta`) computation, respectively. We report cumulative time taken (excluding fetch time) over varying version counts; 2 Spark workers were used for Wikipedia dataset.

respect to the graph updates. This can be seen for a simple node label counting task in Figure 5.10. the benefits due to the incremental (`NodeComputeDelta` operator) computation over a version-based computation (`NodeComputeTemporal` operator) can be seen in Figure 5.15.

5.5 Summary

In this chapter, we addressed two important systemic issues in enabling large scale temporal graph analytics. First, we introduced a data structure called GraphPool, to compactly hold multiple graphs in memory, exploiting temporal overlap between different snapshots, at the same time providing direct access to the snapshots in order to run simultaneous analytical tasks. Additionally, it facilitates materialization or pre-fetching of graph snapshots to speed-up snapshot retrieval from on-disk storage indexes such as DeltaGraph. We demonstrated its efficacy using HiNGE, a system for visual analytics. Second, we discussed the issues in scaling temporal graph analytics to a cluster computing frame-

work through a background survey of various graph analysis routines, graph processing frameworks, etc. We then presented, Temporal graph Analytics Framework (TAF), which proposes appropriate abstractions of operands (such as `SoN` and `SoTS`) and operators (such as `Timeslice` and `NodeComputeDelta`) to enable a variety of temporal analysis tasks over Apache Spark, a state of the art cluster computing framework.

Chapter 6: Conclusion

Graph analytics are increasingly considered crucial in obtaining insights about how interconnected entities behave, how information spreads, what are the most influential entities, and many other characteristics of the domain. Analyzing the history of how a graph evolved 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 dissertation, we presented a novel index called DeltaGraph, that enables persistent compact storage of very large historical graph traces, supporting efficient reconstruction of past states of the graph. It is a highly flexible, tunable and extensible data structure that can be configured to work desirably for various kinds of historical graphs. Building upon DeltaGraph, we presented Temporal Graph Index that supports a wide range of retrieval queries such as node or neighborhood versions, along with snapshots. It is designed to reside in a cloud or cluster, ensuring high scalability and elasticity, which is ideal for very large graphs. We also presented a distributed Temporal Graph Analytics Framework, built on top of Apache Spark, that allows analysts to quickly write complex temporal analysis tasks using a set of novel temporal graph operators. Our experiments show that our temporal indexing technique exhibits very efficient retrieval performance across a wide range of

queries, and can effectively exploit the available parallelism in a distributed setting. Also, our analytical framework is effective in expressing a variety of temporal analysis tasks that can be executed in a parallel manner that is highly scalable.

The work presented in this dissertation will benefit a number of people and impact research and business in different areas. Its primary impact will be in the area of data analytics in the form of a foundation for an appropriate ecosystem of tools that enables temporal graph analytics at scale, with efficiency and ease. Data analytics in turn finds application across several fields. Researchers in biology, sociology, linguistics, and other fields are increasingly adopting data analytics as a power tool to gather insights into large volumes of data to formulate and validate hypotheses. Temporal data with interconnection structures is omnipresent and is a potential source of groundbreaking discoveries in medicine, various branches of science and humanities. We believe that our work presented in this dissertation enables and accelerates the process of data analysis for a number of research works. Several businesses, most prominently in finance, technology, retail and telecommunications are increasingly relying on analyzing massive quantities of consumer data to gain insights for competitive advantage through strategizing, consumer targeting, and improving the quality of their products or services. Temporal analysis of interactions between customers, products and other entities in such large datasets will directly benefit from the techniques presented in this dissertation.

Many of the concepts developed in the course of this research have significance that transcends historical graph data management. Snapshot retrieval proposed through DeltaGraph is oblivious of the underlying graph structure of the data and works well for the model of an evolving set, which can capture relational data amongst other types. In

fact, our experiments proved that DeltaGraph outperforms prior approaches for snapshot retrieval in relational databases. Similarly, the concept of overlaying multiple versions of a graph in-memory to alleviate redundancy using GraphPool also generalizes well beyond graphs. It can perhaps be used in analytical engines for non-graph snapshot-based analysis as well. In the TGI, we used a careful partitioning to handle the temporal and topological skew (non-uniform density). Many non-network datasets exhibit similar skewness patterns and the strategies of partitioning in TGI can be extended to those datasets. Finally, in TAF, while the data modeling is specific to temporal graphs, many of the operators are applicable to non-graph data as well.

Along with the artifacts in form of data structures, algorithms, and tools presented in this dissertation, we also discuss our contribution in the form of certain observations and insights (Section 6.1). We also talk about associated topics of research interest in graph data management and analytics (Section 6.2).

6.1 Insights

In this section, we share some of the key insights about temporal data management, learnt during the course of this work.

- Correct handling of the "temporal consistency" is the key is to tractable storage, retrieval and processing of temporal graph data. The change in a temporal graph can be recorded in several ways. Couple of such examples are the Copy and Log approaches. While they correctly record the entire history and provide support for snapshots, Copy is infeasible for storage, and Log for querying large graphs

that change frequently. DeltaGraph on the other hand performs both the tasks in a feasible and efficient manner. The core reason behind this is the abstraction of temporally consistent portions of the graph through differential functions. This ensures fairly low amount of redundancy but at the same time a direct availability of all portions of a graph. The style of abstraction of the temporally redundant part, i.e., the definition of a differential function, is also the key to control other aspects of the DeltaGraph performance such as latency curves. In other data structures such as GraphPool, SoN, and TGI as well, the choice of handling of the temporal consistency or redundancy is crucial to the success of their performance.

- Time is an orthogonal dimension to that of the graph. Over years, representations of temporal data have treated the quantity of time as a mere attribute. However, for temporal analysis tasks, it is imperative that time be treated as a first-class citizen in both logical and physical modeling of data. A data management system expected to support such analyses requires operations that hinge upon the temporal attribute, such as performing temporal aggregation, temporal joins, temporal filtering and so on; these are doable in a principled (and hence efficient) manner only if the centrality of time is established.

- There is no "one size fits all" solution to problems such as temporal graph indexing. A solution ideal for snapshot retrieval can not be ideal for node-version retrieval. At the core, there exists a natural trade-off when it comes to the appropriate scale of temporal aggregation (or segmentation), and for that matter, partitioning granularity (or clustering) as well. A good solution is to design a system that easily general-

izes to specific scenarios based on configuration of parameters. In our experience, such a system also worked well for a given mixed query load. Also, even in the case of configuration for the best case performance for one type, it still performs reasonably well for the other types of queries. If the users so desire, in the case of optimal performance requirement for multiple query times at once, they may decide to create multiple indexes with different specifications.

6.2 Future Directions

In this section, we discuss some of the interesting and potentially impactful problems in graph analytics besides historical graph data management.

6.2.1 Unified Temporal Graph Processing Framework

Temporal graph analytics can be seen in two realms – historical and streaming. While historical analytics is based upon analyzing past data in large volumes using indexes, the streaming scenario is more about quick insights in a limited context of the current updates and certain running (ongoing) summary statistics. While set up differently, both have various technical commonalities as well as shared objectives. The most prominent goal common to both is perhaps making future predictions on analytical insights. To date, there is no work which harnesses the combined power of analytics from historical and streaming graphs. We believe that a system that integrates the data management aspects of the past (historical) and present (streaming) in graphs, is perhaps the best for future (predictive analytics) insights. Data management and query estimation for streaming graphs is an ac-

tive topic of research, and some background on it can be found in Section 5.2. There has been some work in combining historical and streaming query processing for non graph datasets. Chandrasekaran et al. [43] propose the use of summaries of historical data along with live streaming data to combine analytical reasoning from both the sources. Reiss et al. [109] propose a system to enhance stream query processing by correlating current events with historical trends in TelegraphCQ. Chandramouli et al. [42] propose a system for temporal (streams) query processing on large historical datasets using batch processing frameworks such as Map-Reduce.

6.2.2 Discovering Graphs of Interest

For a user interested in performing graph analysis on the entities in a relational dataset, constructing meaningful graphs requires careful examination of the relational schema, as well as the data itself. Seemingly natural interconnection structures can turn out to be too sparse or too dense or too disconnected to lead to meaningful insights. Manually trying out different possibilities is a time-consuming process, and may miss out on insightful graphs. This issue occurs in temporal analysis as well; in addition to the graphical structure interpretation of attributes from a relational dataset, one or more timestamp fields define the temporality of the network changes. System logs, social media content, telephone call logs, scientific observational datasets are few examples of time annotated datasets that lead to several temporal graph interpretations and make the datasets candidates for temporal graph analysis. To aid the user in this process, we are building a *graph enumeration framework* that performs automated discovery of graphs in a dataset

using a collection of universal rules. We briefly sketch the framework here; we base it on non-temporal graphs for simplicity of explanation but the formulation naturally extends to the temporal dimension. In a normalized database, we consider all *primary keys* to be candidate node sets. For example, in the DBLP dataset, the set of *author-ids* is a candidate node set, and so is the set of *publication-ids*. The process of graph discovery hinges upon finding possible connections for a node set with another (including itself). These connections can be found through the data description constraints such as foreign keys, functional dependencies and attribute co-memberships for a table. More concretely, the problem of discovering homogeneous or bi-partite graphs (i.e., over one or two node sets) can be formalized as follows. Let $\mathcal{N}$ denote the set of primary keys, and $\mathcal{N}'$ denote the set of all remaining attributes in the schema. Let G denote a graph (called *schema graph*) over $\mathcal{N} \cup \mathcal{N}'$, where an edge indicates either a foreign key relationship between a primary key and another attribute in a different table, or a functional dependency between a primary key and another attribute in the same table. A potential homogeneous or bi-partite graph in this dataset can be described as a pair $(X, Y), X, Y \in \mathcal{N}$ (X and Y may be identical), and a path p connecting the two in the schema graph G (the path may traverse the same edge twice, and may visit a node multiple times). A larger set of graphs can be enumerated by adding rules that generate filtering conditions or aggregate operations using the schema graph. Our current implementation supports enumerating graphs without those, and we are working on developing a comprehensive framework for supporting those types of rules.

Besides discovering useful graphs through enumeration, it is also helpful to be able to conveniently extract and analyze specific graphs of interest without going through a

cumbersome and costly process of writing and deploying code. In our recent work, we describe a system called GRAPHGEN [139] that enables users to declaratively specify graph extraction tasks over relational databases, to visually explore the extracted graphs, and to write and execute graph algorithms over them. It demonstrates how unifying the extraction tasks and the graph algorithms enables significant optimizations that would not be possible otherwise.

6.2.3 A Framework for Finding Temporal Patterns in Evolving Graphs

Subgraph pattern matching is a fundamental and powerful operations in graph mining. It captures a variety of search patterns using the notions of subgraph isomorphism and bounded simulation amongst others. In recent years, there has been some work on incremental graph pattern matching which processes updates to a graph to find newer matches. However, the notion of the subgraph pattern itself has largely been restricted to a static sense. We believe that patterns themselves that express a dynamic component, either through temporal constraints or the order of occurrence of events, can greatly enhance the expressivity of graph analytical queries. On the other hand, there is also a lack of principled framework for subgraph pattern matching in historical graphs in terms of decisions on separating what and how to index from what to process post index retrieval, and so on. A general framework to efficiently do a variety temporal pattern matching in graphs seems an interesting and useful area to explore.

6.2.4 Shared Computation Across Graph Versions

While we have discussed at length about the shared storage and retrieval of overlapping graph components between different graph snapshots or versions, utilizing shared computation for shared data across different graphs is also an interesting direction we would like to explore. We would like to point the reader to some of the background work by Ren et al. [110] and Xie et al. [138].

6.2.5 Estimating Changes in Graph Properties

For a graph property, $f()$, studying the correlation between the change in a graph ΔG and the corresponding change in $f(\Delta G)$ would offer better insights into the selection of impactful timepoints. That in turn would make the computation for evolution-based or comparison-oriented tasks more efficient. The naive way of analyzing the evolution of a graph quantity is to compute it at each time point of change in the graph, or alternatively at uniformly separated timepoints. More generally, estimators for the change in a complex quantity, $f(\Delta G)$ based on the knowledge of the change in simpler quantities, $h_1(\Delta G), h_2(\Delta G) \ldots$ can be helpful if the latter are historically indexing and potentially useful for predicting important timepoints of change in $f(\Delta G)$ at query time.

Bibliography

[1] Apache Giraph. http://giraph.apache.org.

[2] Apache Hadoop. https://hadoop.apache.org/.

[3] Apache HBase. http://http://hbase.apache.org/.

[4] Blueprints. http://github.com/tinkerpop/blueprints.

[5] Cassovary. https://github.com/twitter/cassovary.

[6] CloudGraph. .http://www.cloudgraph.com/.

[7] Furnace. http://github.com/tinkerpop/furnace/wiki.

[8] Gremlin. http://github.com/tinkerpop/gremlin/wiki.

[9] Kyoto cabinet. http://fallabs.com/kyotocabinet.

[10] Neo4j. http://www.neo4j.org.

[11] Redis Graph. https://github.com/tblobaum/redis-graph/.

[12] SAIL. https://github.com/tinkerpop/blueprints/wiki/Sail-Implementation.

[13] SPARQL. http://www.w3.org/TR/rdf-sparql-query.

[14] Tinker graph. https://github.com/tinkerpop/blueprints/wiki/TinkerGraph.

[15] Titan: Distributed graph database. http://thinkaurelius.github.io/titan/.

[16] VertexDB. http://www.dekorte.com/projects/opensource/vertexdb/.

[17] Jans Aasman. Allegro graph: RDF triple database. Technical report, Franz Incorporated, 2006.

[18] Charu C Aggarwal and Haixun Wang. Graph data management and mining: A survey of algorithms and applications. In *Managing and Mining Graph Data*, pages 13–68. Springer, 2010.

[19] Charu C. Aggarwal and Haixun Wang. *Managing and Mining Graph Data*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[20] Charu C Aggarwal, Yuchen Zhao, and Philip S Yu. Outlier detection in graph streams. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 399–409. IEEE, 2011.

[21] Jae-wook Ahn, Catherine Plaisant, and Ben Shneiderman. A task taxonomy for network evolution analysis. *IEEE Transactions on Visualization and Computer Graphics*, 2014.

[22] Jae-wook Ahn, Meirav Taieb-Maimon, Awalin Sopan, Catherine Plaisant, and Ben Shneiderman. Temporal visualization of social network dynamics: Prototypes for nation of neighbors. In *Social computing, behavioral-cultural modeling and prediction*, pages 309–316. Springer, 2011.

[23] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM Symposium on Principles of Database Systems*, pages 5–14, 2012.

[24] Mohammed Al-Kateb, Alain Crolotte, Ahmad Ghazal, and Linda Rose. Adding a temporal dimension to the TPC-H benchmark. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 51–59. 2013.

[25] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 560–569. IEEE, 1996.

[26] Sitaram Asur, Srinivasan Parthasarathy, and Duygu Ucar. An event-based framework for characterizing the evolutionary behavior of interaction graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3(4):16, 2009.

[27] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2010.

[28] Eytan Bakshy, Itamar Rosenn, Cameron Marlow, and Lada Adamic. The role of social networks in information diffusion. In *Proceedings of the 21st international conference on World Wide Web*, pages 519–528. ACM, 2012.

[29] Alain Barrat, Marc Barthelemy, and Alessandro Vespignani. *Dynamical processes on complex networks*. Cambridge University Press Cambridge, 2008.

[30] Mathieu Bastian, Sebastien Heymann, Mathieu Jacomy, et al. Gephi: an open source software for exploring and manipulating networks. *Proceedings of the 3rd International AAAI Conference on Weblogs and Social Media (ICWSM)*, 8:361–362, 2009.

[31] Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. The state of the art in visualizing dynamic graphs. In *Proceedings of the Eurographics Conference on Visualization – State of The Art Reports*, 2014.

[32] Tanya Y Berger-Wolf and Jared Saia. A framework for analysis of dynamic social networks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 523–528. ACM, 2006.

[33] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. Principles of dataset versioning: Exploring the recreation/storage

tradeoff. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, 2015.

[34] Gabriele Blankenagel and Ralf Hartmut Gueting. External segment trees. *Algorithmica*, 12(6):498–532, 1994.

[35] Michael Böhlen, Johann Gamper, and Christian S Jensen. Multi-dimensional aggregation for temporal data. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT)*, pages 257–275, 2006.

[36] Azad Bolour, Tera Lougenia Anderson, Luc J Dekeyser, and Harry K. T. Wong. The role of time in information processing: a survey. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1982.

[37] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[38] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. *ACM Transactions on Database Systems (TODS)*, 29(1):2–42, 2004.

[39] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management (CloudDB)*, 2012.

[40] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. *Knowledge and Information Systems*, pages 1–29, 2013.

[41] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: laws, tools, and case studies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 7(1):1–207, 2012.

[42] Badrish Chandramouli, Jonathan Goldstein, and Songyun Duan. Temporal analytics on big data for web advertising. In *Proceedings of IEEE 28th International Conference on Data Engineering (ICDE)*, pages 90–101, 2012.

[43] Sirish Chandrasekaran and Michael Franklin. Remembrance of streams past: overload-sensitive management of archived streams. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 348–359. VLDB Endowment, 2004.

[44] Tain-Jy Chen. Network resources for internationalization: The case of Taiwan's electronics firms. *Journal of Management Studies*, 40(5):1107–1130, 2003.

[45] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *In Proceedings of the 7th ACM european conference on Computer Systems (EUROSYS)*, 2012.

[46] Chris Clarke. Using mongodb as a graph database. `http://engineering.talis.com/articles/using-mongodb-as-graph-db/`.

[47] Chris Date, Hugh Darwen, and Nikos Lorentzos. *Temporal data and the relational model*. Elsevier, 2002.

[48] Prasanna Desikan, Nishith Pathak, Jaideep Srivastava, and Vipin Kumar. Incremental page rank computation on evolving graphs. In *ACM Special interest tracks and posters of at WWW*, 2005.

[49] David Eisenberg, Edward M Marcotte, Ioannis Xenarios, and Todd O Yeates. Protein function in the post-genomic era. *Nature*, 405(6788):823–826, 2000.

[50] Cesim Erten, Philip J Harding, Stephen G Kobourov, Kevin Wampler, and Gary Yee. Exploring the computing literature using temporal graph visualization. In *Electronic Imaging*, pages 45–56. International Society for Optics and Photonics, 2004.

[51] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 38(3):18, 2013.

[52] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 556–567. IEEE, 2014.

[53] Bugra Gedik and Rajesh Bordawekar. Disk-based management of interaction graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(11):2689–2702, 2014.

[54] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems (TODS)*, 21(3), 1996.

[55] Amine Ghrab, Sabri Skhiri, Salim Jouili, and Esteban Zimányi. An analytics-aware conceptual model for evolving graphs. In *Data Warehousing and Knowledge Discovery*. Springer, 2013.

[56] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[57] Fabio Grandi. T-SPARQL: A TSQL2-like temporal query language for RDF. In *Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems*, 2010.

[58] Derek Greene, Donal Doyle, and Padraig Cunningham. Tracking the evolution of communities in dynamic social networks. In *Proceedings of the 2010 international conference on advances in social networks analysis and mining (ASONAM)*, pages 176–183. IEEE, 2010.

[59] Thilo Gross, Carlos J Dommar D'Lima, and Bernd Blasius. Epidemic dynamics on an adaptive network. *Physical Review Letters*, 96(20):208701, 2006.

[60] Ranjay Gulati and Martin Gargiulo. Where do interorganizational networks come from? *American Journal of Sociology*, 104:177–231, 1999.

[61] Jing-Dong J Han, Nicolas Bertin, Tong Hao, Debra S Goldberg, Gabriel F Berriz, Lan V Zhang, Denis Dupuy, Albertha JM Walhout, Michael E Cusick, Frederick P

Roth, et al. Evidence for dynamically organized modularity in the yeast protein–protein interaction network. *Nature*, 430(6995):88–93, 2004.

[62] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 2003.

[63] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *ACM European Conference on Computer Systems*, 2014.

[64] Derek Hansen, Ben Shneiderman, and Marc A Smith. *Analyzing social media networks with NodeXL: Insights from a connected world*. Morgan Kaufmann, 2010.

[65] Huahai He and Ambuj K Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on management of data*, pages 405–418. ACM, 2008.

[66] Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[67] Wenyu Huo and Vassilis J Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2014.

[68] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. GBase: a scalable and general graph management system. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011.

[69] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Ninth IEEE International Conference on Data Mining*, pages 229–238. IEEE, 2009.

[70] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In *ACM SIGMOD International Conference on Management of Data*, 2013.

[71] Matt Kaufmann, Amin Amiri Manjili, Stefan Hildenbrand, Donald Kossmann, and Andreas Tonder. Time travel in column stores. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2013.

[72] Natalie Kerracher, Jessie Kennedy, and Kevin Chalmers. A task taxonomy for temporal graph visualisation. 2015.

[73] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2013.

[74] Udayan Khurana and Amol Deshpande. HiNGE : Enabling temporal network analytics at scale. *ACM SIGMOD International Conference on Management of Data*, 2013.

[75] Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. *Under Submission*, 2015.

[76] Udayan Khurana, Viet-An Nguyen, Hsueh-Chien Cheng, Jae-wook Ahn, Xi Chen, and Ben Shneiderman. Visual analysis of temporal trends in social networks using edge color coding and metric timelines. In *Proceedings of the IEEE Third International Conference on Social Computing (SocialCom)*, pages 549–554. IEEE, 2011.

[77] Georgia Koloniari and Evaggelia Pitoura. Partial view selection for evolving social graphs. In *ACM First International Workshop on Graph Data Management Experiences and Systems*, page 9, 2013.

[78] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15(2):141–145, 1981.

[79] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *Link mining: models, algorithms, and applications*, pages 337–357. Springer, 2010.

[80] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[81] Alan G Labouseur, Jeremy Birnbaum, Paul W Olsen Jr, Sean R Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The g* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, pages 1–36, 2014.

[82] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[83] Nicole Lam and Raymond K Wong. A fast index for XML document version management. In *Web Technologies and Applications*, pages 71–82. Springer, 2003.

[84] Kristina Lerman and Rumi Ghosh. Information contagion: An empirical study of the spread of news on digg and twitter social networks. In *Proceedings of the Fourth International AAAI Conference on Weblogs and Social Media*, volume 10, pages 90–97, 2010.

[85] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.

[86] Jure Leskovec and Rok Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014.

[87] David Lomet, Roger Barga, Mohamed F Mokbel, and German Shegalov. Transaction time support inside a database engine. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2006.

[88] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2012.

[89] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2015.

[90] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[91] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[92] Ian A McCulloh and Kathleen M Carley. Social network change detection. Technical report, Center for the Computational Analysis, 2008.

[93] Andrew McGregor. Graph stream algorithms: A survey. *ACM SIGMOD Rec.*, 43(1):9–20, May 2014.

[94] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system

for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, December 2015.

[95] Boris Motik. Representing and querying validity time in RDF and OWL: A logic-based approach. volume 12, pages 3–21. Elsevier, 2012.

[96] Saket Navlakha and Carl Kingsford. Network archaeology: uncovering ancient networks from present-day interactions. *PLoS Computational Biology*, 7(4), 2011.

[97] Vit Niennattrakul, Pongsakorn Ruengronghirunya, and Chotirat Ann Ratanamahatana. Exact indexing for massive time series databases under time warping distance. *Data Mining and Knowledge Discovery*, 21(3):509–541, 2010.

[98] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[99] Joshua O'Madadhain, Danyel Fisher, Padhraic Smyth, Scott White, and Yan-Biao Boey. Analysis and visualization of network data using JUNG. *Journal of Statistical Software*, 10(2):1–35, 2005.

[100] Gultekin Özsoyoğlu and Richard T Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[101] Gergely Palla, Albert-László Barabási, and Tamás Vicsek. Quantifying social group evolution. *Nature*, 446(7136):664–667, 2007.

[102] Christopher R. Palmer, Phillip B. Gibbons, and Christos Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 81–90. ACM, 2002.

[103] Raj Kumar Pan and Jari Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 2011.

[104] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

[105] Teresa M Przytycka, Mona Singh, and Donna K Slonim. Toward the dynamic interactome: it's about time. *Briefings in bioinformatics*, page bbp057, 2010.

[106] Andrea Pugliese, Octavian Udrea, and VS Subrahmanian. Scaling RDF with time. In *Proceedings of the 17th international conference on World Wide Web*, pages 605–614. ACM, 2008.

[107] Josep M Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: scaling online social networks. In *ACM SIGCOMM Computer Communication Review*, 2011.

[108] Ravi Rajamani. Oracle total recall / flashback data archive. Technical report, Oracle, 2007.

[109] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M Hellerstein. Enabling real-time querying of live and historical stream data. In

Scientific and Statistical Database Management, 2007. SSBDM'07. 19th International Conference on, pages 28–28. IEEE, 2007.

[110] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historial evolving graph sequences. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2011.

[111] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. Technical Report 1039, Stanford University, 2012.

[112] B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2), 1999.

[113] Cynthia M Saracco, Matthias Nicola, and Lenisha Gandhi. A matter of time: Temporal data management in DB2 10. Technical report, IBM, 2012.

[114] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. *Journal of the ACM (JACM)*, 58(3):13, 2011.

[115] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2012.

[116] Jiwon Seo, Sini Guo, and Monica S Lam. Socialite: Datalog extensions for efficient social network analysis. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2013.

[117] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD International Conference on Management of Data*, 2013.

[118] Dennis Shasha, Jason TL Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 39–52. ACM, 2002.

[119] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 236–246, 1985.

[120] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[121] John Snow. *On the mode of communication of cholera*. John Churchill, 1855.

[122] Emad Soroush and Magdalena Balazinska. Time travel in a scientific array database. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2013.

[123] Lei Tang, Huan Liu, Jianping Zhang, and Zohreh Nazeri. Community evolution in dynamic multi-mode networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 677–685. ACM, 2008.

[124] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (editors). *Temporal Databases: Theory, Design, and Implementation*. Benjamin Cummings Publishing Co., 1993.

[125] Chayant Tantipathananandh, Tanya Berger-Wolf, and David Kempe. A framework for community identification in dynamic social networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 717–726. ACM, 2007.

[126] Jonas Tappolet and Abraham Bernstein. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *The Semantic Web: Research and Applications*, pages 308–322. Springer, 2009.

[127] Ian W Taylor, Rune Linding, David Warde-Farley, Yongmei Liu, Catia Pesquita, Daniel Faria, Shelley Bull, Tony Pawson, Quaid Morris, and Jeffrey L Wrana. Dynamic modularity in protein interaction networks predicts breast cancer outcome. *Nature biotechnology*, 27(2):199–204, 2009.

[128] Michael Taylor, Timothy J Taylor, and Istvan Z Kiss. Epidemic threshold and control in a dynamic network. *Physical Review E*, 85(1):016103, 2012.

[129] K Tuncay Tekle, Michael Gorbovitski, and Yanhong A Liu. Graph queries through datalog optimizations. In *Proceedings of the 12th International ACM Symposium on Principles and Practice of Declarative Programming (SIGPLAN)*, pages 25–34. ACM, 2010.

[130] Claudio Tesoriero. *Getting Started with OrientDB*. Packt Publishing Ltd, 2013.

[131] Masashi Toyoda and Masaru Kitsuregawa. A system for visualizing and analyzing the evolution of the web with a time series of graphs. In *Proceedings of the sixteenth ACM conference on Hypertext and hypermedia*, pages 151–160. ACM, 2005.

[132] Vassilis J Tsotras and Nickolas Kangelaris. The snapshot index: an I/O- optimal access method for timeslice queries. *Information Systems*, 20(3):237–260, 1995.

[133] Gergely Varró, Dániel Varró, and Andy Schürr. Incremental graph pattern matching: Data structures and initial experiments. *Electronic Communications of the EASST*, 4, 2006.

[134] Erik Volz and Lauren Ancel Meyers. Epidemic thresholds in dynamic contact networks. *Journal of the Royal Society Interface*, 6(32):233–241, 2009.

[135] Changliang Wang and Lei Chen. Continuous subgraph pattern search over graph streams. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 393–404. IEEE, 2009.

[136] Xiaoyue Wang, Abdullah Mueen, Hui Ding, Goce Trajcevski, Peter Scheuermann, and Eamonn Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26(2):275–309, 2013.

[137] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[138] Wenlei Xie, Yuanyuan Tian, Yannis Sismanis, Andrey Balmin, and Peter J Haas. Dynamic interaction graphs with probabilistic edge decay.

[139] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. Graphgen: Exploring interesting graphs in relational data. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, 2015.

[140] Kevin S Xu, Mark Kliger, and Alfred O Hero III. Visualizing the temporal evolution of dynamic networks. In *Proceedings of the 9th Workshop on Mining and Learning with Graphs*. 2011.

[141] Ji Soo Yi, Niklas Elmqvist, and Seungyoon Lee. TimeMatrix: Analyzing temporal social networks using interactive matrix-based visualizations. *International Journal of Human–Computer Interaction*, 26(11-12):1031–1051, 2010.

[142] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.