A query-oriented approach to graph series distribution and replication: dissertation

Alan G. Labouseur
University at Albany, State University of New York, alan@labouseur.com

The University at Albany community has made this article openly available.
Please share how this access benefits you.

Follow this and additional works at: https://scholarsarchive.library.albany.edu/legacy-etd

Part of the Computer Sciences Commons

Recommended Citation
https://scholarsarchive.library.albany.edu/legacy-etd/1169

This Dissertation is brought to you for free and open access by the The Graduate School at Scholars Archive. It has been accepted for inclusion in Legacy Theses & Dissertations (2009 - 2024) by an authorized administrator of Scholars Archive.
Please see Terms of Use. For more information, please contact scholarsarchive@albany.edu.
A QUERY-ORIENTED APPROACH TO GRAPH SERIES DISTRIBUTION AND REPLICATION

DISSERTATION

by

Alan G. Labouseur

A Dissertation
Submitted to the University at Albany, State University of New York
in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy

College of Computing and Information
Department of Computer Science
University at Albany
Albany, New York
2014
To my lovely wife, Ruobing, and our beautiful boy, Max.
ABSTRACT

Most real-world networks evolve over time. We are surrounded by them: social networks, citation networks, transportation networks, the Web, and more. These evolving, dynamic networks can be modeled as series of graphs (graph snapshots) that represent those networks at different points in time. While many single-graph systems are available today, they lack support for efficiently managing series of large graph snapshots. Our G* system enables efficient storage and querying of these graph snapshots by taking advantage of their commonalities.

In extending G* for scalable and robust operation, we found the classic challenges of data distribution and replication to be imbued with renewed significance given continuously generated graph snapshots. If multiple graph snapshots are commonly queried together, traditional techniques that distribute data over all servers or create identical data replicas may result in inefficient query execution.

This dissertation presents our data distribution techniques, which adjust the set of worker servers for storing each graph snapshot in a manner optimized for popular queries in order to accelerate graph query processing. We also present our data replication approach that maintains each snapshot replica on a different number of workers, making available the most efficient replica configurations for different types of queries.
ACKNOWLEDGMENTS

I would like to express my deepest appreciation and gratitude to my advisor, Prof. Jeong-Hyon Hwang, for his guidance and mentorship. This work would not exist without the benefit of his vast intelligence, clear thought, cautious mind, and wonderfully good-natured companionship during countless early-morning conversations. There is nobody on whose team I would rather be than his. Earning this degree is nice, but earning his friendship has been the real honor of my journey.

I would like to thank Prof. S. S. Ravi for his instruction and guidance, and for his willingness to apply his thorough and analytical mind to improving my work. I would also like to thank Prof. Feng Chen for his helpful advice, ingenious ideas, and contributions to future research opportunities.

While writing this dissertation I collaborated with many colleagues for whom I have great regard. I wish to extend warm thanks to the members of the G* Research Group, including Jeremy Birnbaum, Paul W. Olsen Jr., Sean R. Spillane, Rohini Vabbalareddy, and Jayadevan Vijayan.

I would like to thank my colleagues at Marist College for their friendship and support: Anne Matheus for years of encouragement and for getting me started on this journey; Ron Coleman and Eitel Lauria for pushing me to pursue this degree and implying dire consequences if I did not; Robert Cannistra for morale-boosting and emotional support; Ben Carle and Carolyn Matheus for advice and insight; Roger Norton for his mentorship and support; and all them for the occasional adult beverage.

I am forever grateful to my parents for their love, support, and encouragement, and for instilling in me the values of hard work, education, and persistence.

More than anything, I appreciate and treasure the many years of love and support from my wife and son. Without their love, none of this would have come to pass. Without their love, none of this would matter.
CREDITS

This dissertation is based on several papers jointly written with Professor Jeong-Hyon Hwang and other members of the G* Research Group including Jeremy Birnbaum, Paul W. Olsen Jr., Sean R. Spillane, Jayadevan Vijayan, and Professor Wook-Shin Han. Chapter 3 is based on our G* DAPD journal article [1] and our 2013 ICDE demonstration paper [2]. Chapters 5 and 6 are based on our 2013 $BD^3$ @ VLDB paper [3] and our 2014 WWW demonstration paper [4], respectively.

The G* project has been supported by the National Science Foundation CAREER award IIS-1149372.
CONTENTS

ABSTRACT .................................................................................. iv

ACKNOWLEDGMENTS ................................................................. v

CREDITS .................................................................................... vi

LIST OF FIGURES ........................................................................ x

LIST OF TABLES ............................................................................ xii

1. Introduction ............................................................................. 1
   1.1 Motivation .......................................................................... 1
   1.2 Contributions ...................................................................... 3
   1.3 Dissertation Overview ...................................................... 4

2. Related Work ........................................................................... 5
   2.1 Previous Graph Systems .................................................... 5
   2.2 Storage .............................................................................. 6
   2.3 Indexing ............................................................................ 7
   2.4 Distribution ....................................................................... 7
   2.5 Replication ........................................................................ 7

3. Background: The G* Graph Series Database ......................... 9
   3.1 Architecture ....................................................................... 9
   3.2 Data Storage ...................................................................... 10
       3.2.1 Limitations of the Relational Model ...................... 10
       3.2.2 Our Nested Data Model ....................................... 11
       3.2.3 Efficient Graph Storage ....................................... 12
       3.2.4 Efficient Graph Series Storage ........................... 13
8.1 Conclusions ..................................................... 78
8.2 Future Directions ............................................... 79
# LIST OF FIGURES

3.1 G* Architecture ................................................. 9
3.2 Organization of a Disk Block ................................. 12
3.3 Efficient storage of sequential graph snapshots .............. 13
3.4 Efficient indexing of sequential graph snapshots .......... 14
3.5 CGI Update Examples ....................................... 15
3.6 Average Degree Query Plan – PGQL ......................... 17
3.7 Distributed Computation of Average Degree .................. 18
3.8 Dynamic Graph Queries - DGQL ............................. 19
3.9 Vertex Degree Computation ................................ 22
5.1 Exchanging Segments ..................................... 36
5.2 Maintaining top-k popular graphs and queries .............. 43
6.1 Replica Segment Placement where $r = 2$ ................... 45
6.2 Replica Selection on the G* Master ........................ 46
7.1 The GRASP simulator: Parameters ........................... 52
7.2 The GRASP simulator: Results ................................ 54
7.3 Crossing edges before and after swapping .................. 56
7.4 Segment placement before and after swapping ............... 57
7.5 Standard Ethernet Switch .................................. 58
7.6 Advanced Ethernet Switch ................................. 59
7.7 Shortest Path-type query on one graph (16 servers) ....... 61
7.8 PageRank-type query on one graph (16 servers) ................. 62
7.9 Shortest Path-type query on first and last graphs (16 servers) .... 63
7.10 PageRank-type query on first and last graphs (16 servers) ......... 64
7.11 Shortest Path-type query on every 4th graph (16 servers) ........ 65
7.12 PageRank-type query on every 4th graph (16 servers) ............ 65
7.13 Shortest Path-type query on all graphs (16 servers) ............... 66
7.14 PageRank-type query on all graphs (16 servers) .................. 66
7.15 Shortest Path-type query on one graph (32 servers) .............. 67
7.16 PageRank-type query on one graph (32 servers) .................. 68
7.17 Shortest Path-type query on every 8th graph (32 servers) ........ 68
7.18 PageRank-type query on every 8th graph (32 servers) ............ 69
7.19 Shortest Path-type query on all graphs (32 servers) ............... 69
7.20 PageRank-type query on all graphs (32 servers) .................. 70
7.21 Shortest Path-type query on one graph (64 servers) .............. 71
7.22 PageRank-type query on one graph (64 servers) .................. 72
7.23 Shortest Path-type query on all graphs (64 servers) ............... 72
7.24 PageRank-type query on all graphs (64 servers) .................. 73
7.25 Shortest Path-type query on one graph (128 servers) ............. 74
7.26 PageRank-type query on all graphs (128 servers) .................. 75
7.27 PageRank-type query on all graphs (256 servers) .................. 76
# LIST OF TABLES

<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Impact of Graph Snapshot Placement</td>
<td>25</td>
</tr>
<tr>
<td>7.1</td>
<td>Preliminary Actual Speedup Results</td>
<td>50</td>
</tr>
<tr>
<td>7.2</td>
<td>Preliminary Impact of Graph Data Distribution</td>
<td>51</td>
</tr>
<tr>
<td>7.3</td>
<td>Key Simulation Assumptions</td>
<td>55</td>
</tr>
<tr>
<td>7.4</td>
<td>Query Scenarios</td>
<td>60</td>
</tr>
<tr>
<td>7.5</td>
<td>Simulation Scenarios and Speedup Results</td>
<td>77</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

In this dissertation, we study graph data distribution, replication, and query performance in the context of continuously generated graph snapshots forming a constantly evolving (or dynamic) series of graphs. We present a new dynamic data distribution approach and technique for query-aware graph series distribution. We also present a new technique for query-aware graph series replication.

Our dynamic data distribution technique swaps segments between pairs of workers to reduce overall expected query time. We show that gradually adjusting the number of servers that store each graph while balancing network overhead and server utilization can maximize overall performance.

Our query-aware graph series replication technique maintains each replica on a different number of servers, making available the most efficient storage configurations for various combinations of queries. To further improve query performance, our data replication technique classifies queries into categories and assigns a replica of each data segment to a worker in a manner optimized for that category.

Finally, we show how all of these improvements can be integrated into our dynamic graph database system, G*.

1.1 Motivation

We are surrounded by constantly evolving networks, including social networks, citation networks, transportation networks, and the Web [5]. We can take periodic snapshots of these networks and model them as graphs (where vertices represent entities and edges represent relationships between entities) in order to analyze their features. These features include the distribution of vertex degrees and clustering coefficients [6], network density [7], triadic closure [8], the size of each connected component [6, 9, 10], the shortest distance between pairs of vertices [7, 11, 12], subgraphs
representing congested regions [13], the centrality or eccentricity of vertices [12, 14],
the relative influence of vertices (PageRank) [15], and others [16, 17, 18, 19].

Understanding these dynamic graphs by considering them as graph snapshots plays a crucial role in many application areas including but not limited to social media and sociopolitical analysis [14, 19, 20], national security, marketing both viral and traditional, transportation [13], network management [11, 21], epidemiology [18], pharmacology, and more. All of these applications require “connecting the dots” among a great many data points, which is one of the main challenges in large dynamic graph management.

While many single-graph systems are available today, they lack support for efficiently managing continuously generated series of graph snapshots. G* is our distributed system for managing series of large graph snapshots representing an evolving network at different points in time [1, 2, 3, 4]. It efficiently stores and queries graph snapshots on multiple servers by taking advantage of commonalities among them. This work builds on G* by distributing data over multiple worker servers to accelerate graph query processing and computation.

Accelerating computation by distributing data over multiple servers has been a popular approach in parallel databases [22] and distributed systems [23]. But these techniques do not take into consideration popular queries or graph structure. Techniques for partitioning individual graphs to facilitate parallel computation have also been developed [24, 25, 26, 27]. However, distributing a series of large graph snapshots over multiple servers raises new challenges. In particular, it is not desirable to use traditional graph partitioning techniques which consider only one graph at a time and incur high overhead given a large number of vertices and edges. Furthermore, simply distributing each snapshot on all servers may not be an appropriate approach. If multiple snapshots are commonly queried together, it is more advantageous to store each snapshot on fewer servers as long as the overall queried data are balanced over all servers. In this way, the system can reduce network overhead (i.e., improve query speed) while benefiting from high degrees of parallelism. To do this, we require techniques that (re)distribute with low overhead (achieved with planning) graph snapshots that are continuously generated and take advantage of the property
that \textit{query execution time} depends on both \textit{the number of snapshots queried} and the \textit{distribution} of those snapshots.

We present a new technique that tackles this challenge by splitting graphs into similarly sized segments and then swapping segments between pairs of servers such that the overall expected query time is reduced. To calculate the expected query time, our technique categorizes previously submitted queries according to their operators while keeping track of the CPU, disk, and network overhead as well as the frequency of each query.

1.2 Contributions

In this dissertation, we make original contributions to several important topics in the field of graph series databases.

- \textbf{Query-Oriented Snapshot Distribution}: We present techniques for query-aware graph series distribution. To the best of our knowledge, G* is the first system to address the above challenge of graph data distribution, replication, and query performance in the context of continuously generated graph snapshots. We are, therefore (again, to the best of our knowledge) the first to present techniques for query-aware graph series distribution. We present a new dynamic data distribution approach where each worker partitions its graph data into \textit{similarly-sized segments} with a certain maximum size (e.g., 10GB) so that that worker can control its load by migrating some segments to other workers.

- \textbf{Query Performance Estimation}: To improve query performance, our dynamic data distribution technique swaps segments between pairs of workers such that the overall expected query time is reduced.

This technique also gradually adjusts the number of servers that store each graph while balancing network overhead and server utilization to maximize overall performance.

- \textbf{Graph Series Replication}: We present techniques for query-aware graph series replication. Our technique tolerates up to $r - 1$ simultaneous server
failures by constructing \( r \) replicas for each graph snapshot and maintaining the availability of these replicas in an environment of continuously updated graph snapshots.

- **Query-Oriented Replica Selection**: To further improve query performance, our data replication technique classifies queries into \( r \) categories (where \( r \) is chosen by the G* system administrator according to worker and space availability as well as their replication sensitivity) and optimizes the distribution of each replica for one of the query categories. Our data replication technique maintains each graph replica on a different number of servers, making available the most efficient storage configurations for various combinations of queries.

- **Integration with Dynamic Graph Management**: With these contributions, G* now can tackle new data distribution, replication, and reliability challenges that arise in the context of managing series of large graphs. Our approach continuously routes incoming messages for updating vertices and edges to appropriate workers with low latency. When a segment becomes full, G* splits that segment into two that are similar in size while maintaining data locality by keeping data accessed together within the same segment.

### 1.3 Dissertation Overview

We begin with a discussion of related work and details about some of the giants upon whose shoulders we are now standing in Chapter 2. We continue with a background discussion of the G* Graph Series Database in Chapter 3 and follow that up with formal statements of the distribution and replication problems in the context of graph series segment distribution and replication (Chapter 4). Details of our graph segment distribution and replication solutions comprise the next two chapters, 5 and 6. Chapter 7 contains within it a discussion of our server cluster evaluation results, details of our simulator, and our simulation results. Finally, we conclude in Chapter 8 with a discussion of future research directions.
In this chapter, we briefly summarize related research, focusing on previous graph systems as well as graph storage, indexing, distribution, and replication. Additional discussions of related work can be found in Sections 1.1, 4.1, and 4.2.

2.1 Previous Graph Systems

Relational database systems can readily store graph data but they require breaking down graph structures into edges recorded in a relation [28]. Therefore, graph analysis, using relational databases, involves costly join operations [28, 29].

Graph systems like Giraph [30], GraphLab [31], GraphChi [32], Microsoft’s Trinity [33], the open source Neo4j [34], and others [35, 36, 37, 38] can perform only one operation on one graph at a time, and as such are not suitable for the graph processing needs presented in this dissertation.

Furthermore, neither relational database systems nor most previous graph processing systems take advantage of the commonalities among graphs in the storage and processing of data.\(^1\) These limitations are analytically explained and experimentally demonstrated in our prior work [1].

Google’s Pregel is a parallel graph processing system [39]. A Pregel program includes a user-defined function that specifies superstep-based tasks for each vertex. Using this function, Pregel can execute a graph algorithm such as PageRank or shortest paths algorithms on a server cluster. Pregel achieves a higher level of scalability compared to other graph processing systems such as Parallel BGL [37] and CGM-Graph [35]. Several open-source versions of Pregel are under active development,

\(^1\)These systems cannot readily take advantage of commonalities among graphs and thereby suffer high space overhead. For example, one may consider using a relation to store edges of a series of graphs. In this case, for an edge contained in 100 snapshots, there will be 100 tuples for that edge, each differentiated by snapshot ID. This incurs high space overhead compared to our system, which supports deduplicated storage as described in Chapter 3.
one of which, Phoebus [40], was compared with G* in our prior work [1]. Many open-source systems, like Pegasus [38] are based on Hadoop [41], an open-source implementation of Google’s MapReduce [23]. Carnegie Mellon’s HADI system [42], is based on Hadoop and capable of analyzing very large graphs but it is specifically designed to compute the radii and the diameter of those graphs, whereas G* is far more general.

Researchers are working on using multi-core hardware for graph processing. Ligra [43] is a lightweight graph processing framework specifically for shared-memory multi-core machines. Like Trinity [33], it’s a memory-only system and therefore limited by memory size, while G* is not. Acolyte [44] is a similar in-memory graph system (on a smaller parallel scale) from Tsinghua University. Other recent parallel graph processing systems include Surfer [36] and Angrapa [45]. In contrast to these and other systems, which process one graph at a time [25, 33, 34, 35, 36, 37, 38, 39, 46, 47], G* efficiently executes sophisticated queries on multiple graph snapshots. G*’s benefits over previous systems are experimentally demonstrated in our prior work [1].

DeltaGraph [48] and GraphChi [49] are promising systems for dynamic graphs but do not directly address the graph data distribution and replication issues considered in this dissertation.

2.2 Storage

Graph compression techniques typically store a single graph by either assigning short encodings to popular vertices [50] or using reference compression. Reference compression refers to an approach that represents an adjacency list $i$ using a bit vector which references a similar adjacency list $j$ and a separate collection of elements needed to construct $i$ from $j$ [51, 52]. These techniques and other previous techniques for compressing graphs [53] and binary relations [54] are not well suited for G*’s target applications. In particular, these compression techniques require reconstructing the original vertices and edges, which would slow down the system operation. G*’s storage and indexing mechanisms do not have these limitations but rather expedite queries on multiple graphs.
2.3 Indexing

Researchers have developed various types of graph indexing techniques. Han et al. provided a comprehensive survey and evaluation studies on indexing techniques for pattern matching queries [55]. Jin et al. have recently presented an efficient indexing technique for reachability queries [56] with detailed comparison to other related techniques. In contrast to these techniques, G*’s indexing approach strives to minimize, with low update overhead, the size of the mapping from the vertex and graph IDs to the corresponding graph data on disk. This technique also enables fast cloning of large graphs and allows G* to process each vertex and its edges once and then share the result across relevant graphs to speed up queries on multiple graphs.

2.4 Distribution

Traditional graph partitioning techniques split a static graph into subgraphs in a manner that minimizes the number of crossing edges [24, 26]. There are also recent graph repartitioning schemes that observe communication patterns and then move vertices to reduce network overhead [25, 27]. In contrast to these methods, our technique dynamically adjusts the number of workers that store each graph snapshot according to the real-time influx of graph data and popular types of queries (Chapter 5).

2.5 Replication

There has been extensive work on data replication that focuses on improving data availability and performance [57, 58, 59]. Traditional replication systems persist multiple copies of data on multiple servers for the sole purpose of remaining operational in the face of partial failures. Researchers have developed techniques for ensuring replica consistency [58] and finding most advantageous replica placement [57]. Stonebraker et al. proposed an approach that stores each database replica differently, optimized for a different query type [59]. While our replication approach has some similarity in terms of high-level ideas, it is substantially different in that it distributes each graph snapshot over a different number of workers to speed up
different categories of queries. Thus we take advantage of different replicas to speed up queries (Chapter 6).
CHAPTER 3
Background: The G* Graph Series Database

This chapter provides an overview of G*’s system architecture (Section 3.1), data storage components (Section 3.2), indexing mechanisms (Section 3.3), query languages (Section 3.4) and query processing facilities (Section 3.5).

3.1 Architecture

G* is a distributed system for managing series of directed graphs representing periodic snapshots of evolving networks. It consists of multiple servers. A server that manages the whole system is called the master.

As Figure 3.1 shows, a query submitted to the master is first transformed by the query parser into a network of operators, which is then converted by the query optimizer into an optimized query execution plan (Figure 3.6 is an example of such a plan). Based on the execution plan, the query coordinator instantiates and executes
operators on other servers (Figure 3.7) by controlling their query execution engines. The graph manager on each server stores and retrieves graph data using the server’s memory and disk. The communication layer enables reliable communication with remote servers. These components are currently implemented in approximately 30,000 lines of Java code.

Finally, the high availability module is tasked with performing functions for distributing graph segments among the servers to improve query performance as well as masking server and network failures. This is the main focus of this dissertation.

3.2 Data Storage

A key requirement in G*’s design and implementation is to succinctly store large graphs by taking advantage of their commonalities. Another important requirement is to effectively utilize both the relatively large storage capacity of disks and the high speed of memory. It is crucial to minimize the number of disk accesses in both data storage and retrieval. For example, if each graph edge is accessed with a 10ms disk seek time, it would take 116 days to access 1 billion edges. This sub-section presents a solution that meets these requirements, beginning with why we have not chosen to use the Relational Model.

3.2.1 Limitations of the Relational Model

G* manages three types of entities: graphs, vertices and edges. G* could adopt the relational data model, using a separate relation for each entity type. In this scenario, however, graph queries would be very expensive because of the number and size of the required joins. For example, to retrieve all vertices and edges that belong to certain types of graphs, we would need join operations among the graph, vertex, and edge relations. Furthermore, for a graph traversal query, we would need another join with the edge relation whenever the distance from the source of the traversal increased. It has been shown that for the traversal-type queries, relational databases can be an order of magnitude or more slower than other systems that are based on non-relational technologies [60].
3.2.2 Our Nested Data Model

To avoid the complications mentioned in Section 3.2.1, G* uses a nested data model to capture the inherent relationships among graphs, vertices, and edges in the following logical schema:

\[ \text{graph}(\text{id}, \text{att}_1, \text{att}_2, \ldots, \{\text{vertex}\}) \]

The absolute path to each graph on G*’s distributed file system is used as the primary key, \text{id}. Each \text{att}_i is a graph attribute and \{\text{vertex}\} is the set of vertices contained in the graph.

For the vertices in the same graph, we use the following logical schema:

\[ \text{vertex}(\text{id}, \text{att}_1, \ldots, \{\text{edge}\}) \]

where \text{id} is an identifier that distinguishes among the vertices in the same graph, each \text{att}_i is a vertex attribute, and \{\text{edge}\} is the set of edges emanating from the vertex identified by \text{id}. Given multiple graphs and their vertices, the primary key for uniquely identifying a vertex is \text{graph.id}, \text{vertex.id}, where \text{graph.id} and \text{vertex.id} are the graph and vertex IDs mentioned above.

For the edges that emanate from the same vertex, we use the following logical schema:

\[ \text{edge}(\text{id}, \text{att}_1, \ldots) \]

where \text{id} is an identifier distinct among the edges based on the IDs of the vertices to which the edges are incident, and each \text{att}_i is an edge attribute. The primary key for uniquely identifying each edge is thus \text{graph.id}, \text{vertex.id}, \text{edge.id}, where \text{graph.id}, \text{vertex.id} and \text{edge.id} are the graph, vertex, and edge IDs mentioned above.
Putting it all together, we can see the entire nested data model:

```
graph(id, att1...n,
    {vertex(graph.id, id, att1...n,
        {edge(vertex.id, graph.id, des_id, att1...n))
    })
)
```

### 3.2.3 Efficient Graph Storage

In a variety of graph applications, the edges of a vertex must be processed together (Section 3.5.1). To minimize the number of disk accesses, G* stores each vertex and its edges within the same logical disk block. All of the data within a disk block is loaded and saved together and the logical disk block size is configurable. (The default size is 256KB in the current implementation.) For each vertex, the vertex ID, attribute values, and all of the outgoing edges are stored on disk (Figure 3.2). For each edge, the ID of the destination vertex and the attribute values of the edge are saved on disk. In Figure 3.2, two versions \( c_1 \) and \( c_2 \) of vertex \( c \) are stored within disk block 10 at indices 0 and 3, respectively. For space efficiency, \( c_1 \) and \( c_2 \) share commonalities. This type of deduplicated storage of complex objects is supported by Java serialization. The above disk locations are represented as “10:0” and “10:3”, respectively. To quickly access disk-resident data, each G* server uses a memory buffer.

![Figure 3.2: Organization of a Disk Block.](image)

Objects are allocated from the end of the block while data about these objects is stored from the beginning of the block.
Figure 3.3: Efficient storage of sequential graph snapshots. Each server efficiently manages a subset of vertices and edges from multiple graphs.

to keep a memory cache of disk blocks for use by its graph manager (Figure 3.1).

3.2.4 Efficient Graph Series Storage

As noted in Section 3.1, G* manages series of directed graphs representing periodic snapshots of evolving networks using multiple servers. It handles undirected graphs by using directed graphs that contain, for each undirected edge, two corresponding directed edges, one in each direction. G* can ingest streaming data from external sources such as Twitter’s Gardenhose [61] or it can import data files. According to such input data, it adds, deletes, and updates vertices and edges and their attributes. G* can also create a series of cumulative graphs by periodically cloning the current graph [1]) and then updating only the new graph according to the new data. Our focus in this dissertation is specifically on querying graphs that correspond to periodic snapshots of an evolving network.

G* assigns a vertex and its outgoing edges to the same server for high data locality. For example, in Figure 3.3(a), server α can access every edge of vertex a without contacting other servers. An update of an edge is therefore handled by the server that stores the source vertex of that edge. The current G* implementation assigns each new vertex to a server based on the hash value of its vertex ID. In
Figure 3.4: Efficient indexing of sequential graph snapshots. Server $\gamma$ at time 3. Vertex $e$ and edge $(c, e)$ having been added in graph $G_2$ at time 2, server $\gamma$ now stores two versions of $c$ ($c_1$ in $G_1$ and $c_2$ with an edge to $e$ in $G_2$) in a deduplicated manner. Now at time 3, with the addition of vertex $f$ and edge $(d, f)$ in graph $G_3$, server $\gamma$ stores two versions of $d$ ($d_1$ in both $G_1$ and $G_2$, and $d_2$ with an edge to $f$ in $G_3$).

In this dissertation, we demonstrate that this balanced, essentially random distribution results in unnecessarily long query times for certain types of queries common to several analytic use cases (Chapter 5).

Each G* server strives to efficiently manage data by taking advantage of commonalities among the graphs. For example, server $\alpha$ in Figure 3.3(a) is assigned vertex $a$ and its outgoing edges which remain the same in graphs $G_1$, $G_2$ and $G_3$. Thus, server $\alpha$ stores vertex $a$ and its edges only once on disk. On the other hand, vertex $d$ obtains a new edge to $f$ in graph $G_3$ (Figure 3.3(c)). In response to this update, server $\gamma$ stores $d_2$, a new version of $d$, which shares commonalities with the previous version, $d_1$, for space efficiency. As this example shows, if a vertex’s attributes or outgoing edges change in a graph, the corresponding server saves a new version of the vertex on disk. If a vertex and its edges are updated multiple times in a graph, only the most recent version is kept.

### 3.3 Indexing

Each G* server maintains an index to quickly find the disk location of a vertex and its edges given relevant vertex and graph IDs. This index also takes advantage of commonalities among the graphs to reduce its footprint. For this reason, we call this the Compact Graph Index (CGI). Specifically, the CGI stores only one (vertex
Figure 3.5: CGI Update Examples. Figures 3.5(a) and 3.5(d) show server $\gamma$’s CGI in Figures 3.3(a) and 3.3(b), respectively.

For $c_2$, $\gamma$’s CGI stores $(c, \text{location}(c))$ only once in a VL map for the combination of $G_2$ and $G_3$ rather than redundantly storing it for each of $G_2$ and $G_3$. This CGI efficiently stores vertex IDs and disk locations whereas all of the attribute values of vertices and edges are saved on disk. Due to its small size, the CGI can be kept fully or mostly in memory, enabling fast lookups and updates. To prevent the CGI from managing too many combinations of graphs, each G* server also automatically groups graphs and separately indexes each group of graphs [1].

As Figure 3.5 shows, the CGI maintains VL pairs in a deduplicated fashion by using VL maps. In the current CGI implementation, VL maps use B+ trees. The size of each VL pair (e.g., 16 bytes for the ID and disk location of a vertex) is in general much smaller than that of the disk resident graph data (e.g., 10 kilobytes of data storing all of the attribute values of a vertex and its edges). Each G* server therefore can usually maintain all or most of its CGI in memory, thus achieving fast data lookup.

The CGI needs to maintain multiple VL maps, one per combination of stored graphs. To iterate over all of the vertices in each graph, the CGI has a root map that
associates each graph ID with all of the relevant VL maps (see the shaded triangle in Figure 3.5(d) that associates \( G_1 \) with VL maps for \( \{G_1\} \) and \( \{G_1, G_2\} \)).

While the CGI has benefits in terms of storage, the update overhead of the CGI increases with more VL maps. As experimentally demonstrated in our prior work [1], the number of VL maps managed by the CGI usually does not increase exponentially with the number of graphs. In particular, given a series of cumulative graphs, the number of VL maps increases at most quadratically. The reason behind this phenomenon is that in graphs \( \{G_i\}_{i=1}^N \), each vertex version is created in some graph \( G_\alpha \) and remains the same in the subsequent graphs until it is superseded by a new version in graph \( G_\omega \). This means that common vertices and edges always belong to graph combinations of the form \( \{G_i\}_{i=\alpha}^{\omega-1} \).

### 3.4 Query Languages

\( G^* \) currently supports two query languages. They are are summarized in Sections 3.4.1 and 3.4.2.

#### 3.4.1 Procedural Graph Query Language

The Procedural Graph Query Language (PGQL) can directly define a network of operators that the master server constructs on \( G^* \) worker servers. PGQL operators fall into two categories: those that operate on graphs, such as the VertexOperator, which retrieves relevant vertices and their edges from disk, and non-graph operators such as the AggregateOperator, which conducts aggregation operations including count, sum, min, max, and avg. Other non-graph operators include UnionOperator, ProjectionOperator, JoinOperator, SortOperator and TopKOperator. For details about and examples of all of these operators, please see our \( G^* \) Operator Reference [62].

The commands for creating operators have the following form:

\[
\text{<op\_name>}@<\text{server\_id} | *> = \\
\text{<op\_type>}([\text{<input\_op\_names>}], \text{<param 1>}, \text{<param 2>}, ...) \\
\]

Each operator creation command constructs a new \text{<op\_type>-type operator} called
Figure 3.6: Average Degree Query Plan – PGQL. Each line specifies (a) the type of operator to create (e.g., `VertexOperator` on line 1) as well as arguments including the operators to connect to in order to obtain input data (e.g., `vertex@local` on line 2 refers to an operator labeled `vertex` on the same server), (b) the servers to create the operator (e.g., `@*` on lines 1-3 and `@alpha` on lines 4-5 indicate operator creation on all servers and on server `alpha`, respectively), and (c) the label assigned to the operator to create (e.g., `vertex` on line 1).

```plaintext
1 vertex@* = VertexOperator([], '/twitter/*');
2 degree@* = DegreeOperator([vertex@local]);
3 count_sum@* = PartialAggregateOperator([degree@local],
    count_sum, degree, graph.id);
4 union@alpha = UnionOperator([count_sum@*]);
5 avg@alpha = AggregateOperator([union@local], avg, graph.id);
```

<op_name> on server <server_id> or all servers (*). Here, <input_op_names> is a list of operators that provide data to the operator(s) to create, and each <param_i> is a parameter needed during operator creation. For example, in Figure 3.6, `vertex@local` (on line 2) refers to the operator named `vertex` on the server on which the operator is constructed, where as `count_sum@*` (on line 3) refers to all operators named `count_sum` on all servers. On the other hand, `union@alpha` (on line 5) refers to the operator named `union` on server `alpha`.

Given such a command, the G* master instructs each relevant server (as denoted in `<op_name>@<server_id | *>`) to instantiate an operator of the specified type (`<op_type>`), which is implemented in advance as a Java class (e.g., `UnionOperator`). Then, each server creates an operator using the constructor that matches the specified operator type and parameters (found through Java reflection) and connects this operator to other operators according to the `<input_op_names>` phrase of the command.

Figure 3.6 shows an example which computes the average vertex degree for each graph located in the `/twitter/` directory on G*’s distributed file system. The `VertexOperator` retrieves all of the vertices from graphs from a given input specifier that match a supplied condition (in Figure 3.6, all of the are graphs located in `/twitter/`). The `DegreeOperator` computes the vertex degree for the given input. The `AggregateOperator` and `PartialAggregateOperator` take input from a
specified source (e.g., from `degree` on line 3 in Figure 3.6) and execute the specified functions on that input based on the specified attributes and then store the results in the specified output attributes.

Figure 3.7 illustrates a network of operators constructed according to the example in Figure 3.6. Figure 3.7 assumes that graphs $G_1$, $G_2$, and $G_3$ are located in the `/twitter/` directory, and further that each server has grouped its vertices and edges based on the graphs that have them in common. The `vertex` and `degree` operators on each server compute the degree of each vertex while associating the result with the IDs of the related graphs. For example, the output of `degree` on server $\alpha$ indicates that the degree of $a$ is 2 in graphs $G_1$, $G_2$, and $G_3$. The `count_sum` operator on each server then computes the count and sum of the received vertex degrees with grouping on `graph.id` (see also lines 3-4 in Figure 3.6). These partial aggregate values computed on each server are merged by the `union` operator on server $\alpha$ and then processed by the `avg` operator, which computes the final result. Conceptually, $G^*$ can support any query language that can be translated into PGQL.
<table>
<thead>
<tr>
<th>Line</th>
<th>SQL Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>select graph.id, avg(degree) -- Q1. average vertex degree</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>from (select graph.id, degree(vertex) as degree</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>from graph('/twitter/*'))</code></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>group by graph.id</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>select graph.id, coeff10*0.1, count(*) -- Q2. clustering coefficient dist.</code></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>from (select graph.id, floor(c_coeff(vertex)*10) as coeff10</code></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><code>from graph('/twitter/*'))</code></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><code>group by graph.id, coeff10</code></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><code>select graph.id, min_dist, count(*) -- Q3. minimum distance distribution</code></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><code>from min_dist(graph('/tree/*'), '1') -- vertices with min_dist</code></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td><code>group by graph.id, min_dist</code></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td><code>select graph.id, comp_size, count(*) -- Q4. component size distribution</code></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td><code>from (select graph.id, comp_id, count(*) as comp_size</code></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td><code>from comp_id(graph('/twitter/*')) -- vertices with comp_id</code></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>`group by graph.id, comp_id)</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td><code>select graph.id, comp_size, count(*) -- Q4. component size distribution</code></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td><code>from (select graph.id, comp_id, count(*) as comp_size</code></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td><code>from comp_id(graph('/twitter/*')) -- vertices with comp_id</code></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>`group by graph.id, comp_size)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.8: Dynamic Graph Queries - DGQL

3.4.2 Declarative Graph Query Language

The second language that G* provides is called Declarative Graph Query Language (DGQL). DGQL is similar to SQL, but closer to OQL [63] in that it enables queries defined upon sets of complex objects (e.g., vertices referencing edges to other vertices).

Figure 3.8 shows example queries that are based on the representative applications mentioned in Section 1.1. These queries compute, for each graph of interest, the average vertex degree (Q1), as well as the distribution of clustering coefficients (Q2), the shortest distances to vertices from vertex '1' (Q3), and the sizes of connected components (Q4).

In Figure 3.8, `degree()` on line 2 and `c_coeff()` on line 7 compute the degree and clustering coefficient of each vertex, respectively. `min_dist()` on line 12 computes the shortest distance from vertex '1' to each vertex for every graph in the '/tree/' directory. `min_dist()` outputs objects that contain the ID of a vertex v and the `min_dist` value (i.e., the shortest distance from vertex '1' to v). `comp_id()` on line 17 finds the connected components for each graph in the '/twitter/' directory.
comp_id() assigns the same component ID to all of the vertices that are within the same component. comp_id() outputs objects which contain the ID of a vertex and the comp_id value (the ID of the component that contains the vertex).

3.5 Query Processing

G*, like traditional database systems, transforms queries into a network of operators that process data in a pipelined fashion. Unlike traditional database systems, G* implements a new query processing framework that executes operators on multiple servers to efficiently process distributed graphs, has an index that associates each vertex with all of the graphs that contain that vertex (Section 3.3), shares computations across relevant graphs, and supports operators specifically for solving graph problems.

Given a query execution plan (expressed as PGQL, as in Figure 3.6), the G* master server constructs a network of operators (Figure 3.7) according to that plan. Each G* server, including the master, interacts with the others using remote method invocations (RMIs) [64]. For example, given command

\[
\text{vertex}@* = \text{VertexOperator}([], '/twitter/*')
\]

(line 1 in Figure 3.6), the master invokes the \text{createOperator()} method on each G* server while passing in (1) the label to assign to that operator (vertex), (2) the type of the operator (\text{VertexOperator}), (3) the operators to connect to for input data (\text{[]}, meaning none in this case), and (4) arguments (the pattern that expresses the graphs to process, '/twitter/*' in this case).

A G* operator, such as degree in Figure 3.7, can obtain data from another operator by receiving an iterator from the operator and then repeatedly calling \text{next()} on that iterator. Just like traditional database management systems, this iterator-based approach is for pipelined transmission and processing of data.

Using our RMI service, the \text{union} operator on server $\alpha$ in Figure 3.7 obtains an iterator for getting data from the remote operator \text{count_sum} on server $\beta$. In this case, the \text{union} operator is given a proxy iterator constructed on server $\alpha$ on behalf
of the original iterator that the `count_sum` operator provides on server β. To help the `union` operator on server α efficiently process data, server β proactively retrieves data using the original iterator from `count_sum` and sends the data to server α. This approach enables pipelined processing.

The current G* implementation supports the graph processing operators discussed below in addition to other operators that are analogous to traditional relational operators, such as selection, projection, aggregate, and join. These operators may directly read disk-resident graph data (e.g., the `vertex` operator in Figure 3.7), receive data streams from other operators (e.g., `degree`, `count_sum`, `union`, `avg` operators), or exchange special summary values with each other to solve a graph problem. Each operator produces a stream of data objects that represent the result (e.g., the ID and degree of each vertex in the case of the `degree` operator).

### 3.5.1 Sharing Computations across Graphs

Each `vertex` operator in Figure 3.7 obtains an iterator from the set of CGIs (Figure 3.5) that cover the graphs being queried (e.g., \{G₁, G₂, G₃\}). Each invocation of `next()` on this iterator provides the disk location that stores a vertex and its edges, as well as the IDs of the graphs that contain them. Based on this input data, the `vertex` operator reads relevant data from disk and produces data objects, each of which represents a vertex, its edges, and the IDs of the graphs that contain them (e.g., \( (a, \ldots, \{G₁, G₂, G₃\}) \)) on server α in Figure 3.7). In G*, the `Vertex` type is used for these objects. If a vertex and its edges do not change across multiple graphs, the `vertex` operator loads them only once from disk and then associates them with the IDs of the graphs that contain them.

G* operators that consume the output stream of the `vertex` operator can naturally share computations across relevant graphs. For example, the `degree` operator on server α in Figure 3.7 computes the degree of vertex \( a \) only once and then incorporates the result (i.e., degree of 2) into the `Vertex` object that represents \( a \) in graphs \( G₁, G₂, \) and \( G₃ \). This change affects only the `Vertex` object in memory and has no influence on the base data on disk. It should be noted that the above operation involves only one disk access to load vertex \( a \) and its edges, as well as one single
```java
public Vertex next() {
    Vertex v = input(0).next();
    v.add("degree", v.degree());
    return v;
}
```

**Figure 3.9: Vertex Degree Computation**

computation of a’s degree while sharing the result across graphs $G_1$, $G_2$, and $G_3$. In other systems that can process only one graph at a time, the same result would require three series of disk access and degree computation, one for each of the three graphs. Our shared computation is more beneficial when vertices and edges remain the same across a larger number of graphs.

Figure 3.9 shows the actual implementation of the `degree` operator. In this implementation, the complexity of dealing with multiple graphs is completely hidden. The reason behind this benefit is that the actual code for computing the degree of a vertex (i.e., counting the number of outgoing edges) does not require any information about the graphs which contain that vertex.

### 3.5.2 Implementing Graph Theoretic Operators

The BSP (Bulk Synchronous Parallel) model has been frequently used in various parallel graph algorithm implementations [36, 37, 39]. This model uses a number of iterations called *supersteps* during which a user-defined custom function is applied to each vertex in parallel. This custom function changes the state variables of a vertex based on the their current values and the messages received during the previous superstep. The overall computation completes when a certain termination condition is met (e.g., no state variable changes for any of the vertices).

In G*, operators that support the BSP model can be implemented by extending the `BSPOperator` class and completing the following method:

- `void compute(Vertex v, Summary<V, F> s)`: carries out a certain task based on vertex `v` and summary `s` for `v`. This method is invoked only when a summary bound to `v` has arrived for the first time, or the summary already associated
with \( v \) is updated with the summaries received during the previous superstep.

We describe our PageRank and SSSP operators are implemented using this method in our prior work [1].

### 3.6 Summary

In this chapter, we described our G* Dynamic Graph Database system in terms of its architecture, data storage abilities, indexing capabilities, query language support, and query processing facilities. It is in this context that we turn our attention to formally defining query-oriented snapshot distribution and replication.
CHAPTER 4

Problem Statements

4.1 Graph Snapshot Distribution

Accelerating computation by distributing data over multiple servers has been a popular approach in parallel databases [22] and distributed systems [23]. Furthermore, techniques for partitioning graphs to facilitate parallel computation have also been developed [24, 25, 26, 27]. However, distributing large graph snapshots over multiple workers raises new challenges. In particular, it is not desirable to use traditional graph partitioning techniques which consider only one graph at a time and incur high overhead given a large number of vertices and edges. Solutions to this problem must (re)distribute with low overhead graph snapshots that are continuously generated and take advantage of the property that query execution time depends on both the number of snapshots queried and the distribution of the graph snapshots as illustrated below.

4.1.1 Example

Consider a scenario where each of 100 similarly-sized graph snapshots contain approximately 1 million vertices and 100 million edges. Assume also that the system consists of one master and 100 workers. Table 4.1 compares two snapshot distribution configurations (or placements): Shared-Nothing, where each of the 100 snapshots is stored on one distinct worker, and Shared-Everything, where each snapshot is uniformly distributed over all of the 100 workers. For each of these configurations, two kinds of PageRank-type queries are executed: Query One (graph) snapshot, and Query All (graph) snapshots. The explanations below are based on our previous preliminary evaluation results (see Section 7.1 for details).

In the case of Shared-Nothing, querying one snapshot using only one worker takes 245 seconds (205 seconds to construct the snapshot from disk and 40 seconds to run 10 iterations of PageRank). Querying all snapshots on all workers in parallel...
<table>
<thead>
<tr>
<th>PageRank Query</th>
<th>Shared-Nothing</th>
<th>Shared-Everything</th>
</tr>
</thead>
<tbody>
<tr>
<td>One snapshot</td>
<td>245 seconds</td>
<td>22 seconds</td>
</tr>
<tr>
<td>All snapshots</td>
<td>245 seconds</td>
<td>2,205 seconds</td>
</tr>
</tbody>
</table>

Table 4.1: Impact of Graph Snapshot Placement

takes the same amount of time. When the *Shared-Everything* configuration is used, querying one snapshot on all workers takes approximately 22 seconds, mainly due to network communications for the edges that cross worker boundaries (the disk I/O and CPU costs correspond to only 205/100 seconds and 40/100 seconds, respectively, due to the distribution of the snapshot over 100 workers). In this configuration, querying 100 snapshots takes 2,205 seconds as the PageRank of each vertex varies across graph snapshots, thereby causing 100 times more message transmissions than the previous case.

This example shows the benefits of different snapshot distribution approaches for different types of queries (e.g., *Shared-Nothing* for queries on all snapshots and *Shared-Everything* for queries on one snapshot).

4.1.2 Formal Definition

Our ultimate goal is to keep track of the popularity of graph snapshots and to optimize the storage/distribution of *unpopular* snapshots for space efficiency and *popular snapshots* for query speed. In this dissertation, we focus on the problem of distributing popular snapshots over workers in a manner that minimizes the execution time of queries on these snapshots. This problem can be formally defined as follows:

**Problem 1. (Snapshot Distribution)** Given a series of graph snapshots \( \{G_i(V_i, E_i) : i = 1, 2, \cdots \} \), \( n \) workers, and a set of queries \( Q \) on the snapshots, find a distribution \( d = \{V_{i,w} : i = 1, 2, \cdots \text{ and } w = 1, 2, \cdots, n\} \) that minimizes \( \sum_{q \in Q} time(q, d) \) where \( V_{i,w} \) denotes the set of vertices from snapshot \( G_i(V_i, E_i) \) which are assigned to worker \( w \), and \( time(q, d) \) represents the execution time of query \( q \in Q \) on the distributed snapshots \( d \) satisfying (1) \( \bigcup_{w=1}^n V_{i,w} = V_i \) (i.e., the parts of a snapshot on all workers cover the original snapshot) and (2) \( V_{i,w} \cap V_{i,w'} = \emptyset \) if \( w \neq w' \) (i.e., workers are assigned disjoint parts of a snapshot).
4.1.3 Definitions for Hardness

An approximation algorithm (or heuristic) for an optimization problem is a polynomial time algorithm that produces a feasible solution (i.e., a solution that satisfies all the specified constraints) for all instances of the problem. However, the solution produced by a heuristic may not be optimal for all instances of the problem.

Given a minimization problem $\Pi$ and a number $\rho \geq 1$, a heuristic $H$ for $\Pi$ is a $\rho$-approximation algorithm if for every instance of $\Pi$, the solution value produced by $H$ is at most $\rho$ times the optimum solution value for that instance. A $\rho$-approximation algorithm is said to provide a performance guarantee of $\rho$.

The following problem, which is known to be NP-complete, will be used in proving a complexity result for the Snapshot Distribution problem.

1-in-3 3SAT with Positive Literals (PL-1-3-3SAT)

Instance: A set $X = \{x_1, x_2, \ldots, x_n\}$ of Boolean variables, a set $C = \{C_1, C_2, \ldots, C_m\}$ of $m$ clauses, where each clause consists of a 3-element subset of $X$.

Question: Is there an assignment of truth values to the variables in $X$ such that exactly one variable is assigned the value True in each clause of $C$?

It is known that PL-1-3-3SAT is NP-complete [65].

4.1.4 The Complexity of Snapshot Distribution

We use the abbreviation SD for the Snapshot Distribution problem. This section establishes the following result.

Theorem 4.1. Problem SD is NP-hard. Moreover, for any $\rho \geq 1$, no polynomial time approximation algorithm can provide a performance guarantee of $\rho$ for SD, unless $P = NP$. 

26
Proof of NP-hardness: We first prove the NP-hardness of SD using a reduction from the PL-1-3-3SAT problem defined above. Given an instance of PL-1-3-3SAT consisting of variable set $X$ and clause set $C$, we produce an instance of SD as follows.

1. The SD instance consists of just one graph $G(V,E)$. The vertex set $V = A \cup B$, where $A = \{a_1, a_2, \ldots, a_n\}$ corresponds to the variable set $X$ and $B = \{b_1, b_2, \ldots, b_m\}$ corresponds the clause set $C$. $G$ is a complete graph; that is, there is an edge between every pair of vertices in $G$.

2. The query set $Q = \{q_1, q_2, \ldots, q_m\}$ consists of $m$ queries, one corresponding to each clause. If clause $C_j = (x_r \lor x_s \lor x_t)$, then query $q_j$ involves the complete graph on the four vertices $b_j, a_r, a_s$ and $a_t$.

3. The number of worker/servers = 2.

4. The time function to evaluate queries is as follows. Consider any query $q_j$ which involves the four vertices $b_j, a_r, a_s$ and $a_t$. If the distribution of the vertices of $G$ is such that $b_j$ and exactly one of $a_r$, $a_s$ and $a_t$ are assigned to Worker 1 and the remaining two are assigned to Worker 2, the time function has the value 1; otherwise, the time function has the value 2.

5. The bound on the total time to evaluate all the $m$ queries is set to $m$.

This completes the construction of the SD problem instance. It can be verified that this construction can be carried out in polynomial time. We now show that there is a solution to the resulting SD instance (i.e., distribution of the vertices of $G$ over the two servers with total query time less than equal to $m$) iff there is a solution to the PL-1-3-3SAT instance.

Suppose there is a solution to the PL-1-3-3SAT instance. Let $X' \subseteq X$ be the set of variables that are set to True in this assignment. Consider the following
distribution of the vertices of $G$ over the two workers: assign all the vertices of $B$ and those vertices of $A$ that correspond to the variables in $X'$ to Worker 1 and the remaining vertices to Worker 2. Since $X$ is a solution to the PL-1-3-3SAT instance, each vertex $b_j$ appears on Worker 1 with exactly one of the other vertices appearing in query $q_j$. Thus, the time to evaluate each query is 1, and since there are $m$ queries, the total time to evaluate all the queries is $m$. In other words, there is a solution to the SD instance.

Now suppose there is a solution to the SD instance. Since the time to evaluate each query is at least 1 and there are $m$ queries, the time to evaluate all the queries is at least $m$. Given that the query evaluation time is at most $m$, it follows that the time is exactly $m$ and that the time to execute each of the queries is exactly 1. By the definition of our time function, it follows that for each query $q_j$ which involves the vertex $b_j$ and three other vertices from $A$, $b_j$ and exactly one of those three vertices are assigned to Worker 1 and the other two vertices in $q_j$ are assigned to Worker 2. Now construct a solution to the PL-1-3-3SAT instance as follows. Let $X'$ be the subset of Boolean variables corresponding to those vertices of $A$ that have been assigned to Worker 1. Assign the value True to the variables in $X'$ and the value False to the remaining variables in $X$. It can be verified that this assignment sets exactly one variable in each clause $C_j$ to True. In other words, this is a solution to the PL-1-3-3SAT instance. This completes the proof that SD is NP-hard.

**Proof of Non-approximability:** Suppose $\mathcal{H}$ is a polynomial time approximation algorithm with a performance guarantee of $\rho \geq 1$. We show that the PL-1-3-3SAT problem can be solved in polynomial time using $\mathcal{H}$. This would contradict the assumption that $P \neq NP$ and establish the non-approximability result.

Given any instance of the PL-1-3-3SAT problem, we use the same construc-
tion as above except that the definition of the *time* function is modified as follows. Consider any query $q_j$ which involves the four vertices $b_j, a_r, a_s$ and $a_t$. If the distribution of the vertices of $G$ is such that $b_j$ and *exactly one* of $a_r, a_s$ and $a_t$ are assigned to Worker 1 and the remaining two are assigned to Worker 2, the *time* function has the value 1; otherwise, the *time* function has the value $m\rho + 1$. We now have the following claim.

**Claim 1:** Approximation algorithm $\mathcal{H}$ produces a distribution with total time at most $m\rho$ iff the PL-1-3-3SAT instance has a solution.

**Proof of Claim 1:** Suppose the PL-1-3-3SAT instance has a solution. As argued above, there is a distribution of $G$ that leads to a total execution time of $m$ for all the $m$ queries. Since $\mathcal{H}$ provides a performance guarantee of $\rho$, the distribution produced by $\mathcal{H}$ must have a total execution time of at most $m\rho$.

Now, suppose there is no solution to the PL-1-3-3SAT instance. Thus, for every assignment of values to the Boolean variables in $X$, there is some clause $C_j$ such that the number of variables in $C_j$ which are set to True is not equal to 1. In other words, no matter how the vertices of $G$ are distributed, there is at least one query $q_j$ such that the vertex $b_j$ of $q_j$ does not appear with exactly one of the other vertices in $q_j$ on Worker 1. Thus, the time to evaluate $q_j$ is $m\rho + 1$, and this completes the proof of Claim 1.

Since $\mathcal{H}$ was assumed to run in polynomial time, the result of Claim 1 contradicts the assumption that $P \neq NP$.

Since SD is an NP-complete problem, brute-force and exhaustive search methods will not be practical for the large graphs that we want to consider. We are going to need a heuristic approach. Our solution to this problem, including our heuristic approach, is presented in Chapter 5.
4.2 Graph Snapshot Replication

There have been various techniques for replicating data to improve availability and access speed [57, 58, 59]. A central data replication challenge in G* is to distribute each replica of a snapshot over a possibly different number of workers to maximize both performance and availability. For each query, the most beneficial replica also needs to be found according to the characteristics of the query (e.g., the number of snapshots queried). If two replicas of a graph snapshot are distributed using the Shared-Nothing and Shared-Everything approaches, queries on a single snapshot should use the Shared-Everything replica configuration rather than the other. In practice, however, each query can access an arbitrary number of graph snapshots (not necessarily one or all), thereby complicating the above challenges. The problem of replicating graph snapshots can be defined as follows:

4.2.1 Formal Definition

**Problem 2. (Snapshot Replication)** Given a series of graph snapshots \( \{G_i(V_i, E_i) : i = 1, 2, \ldots \} \), the degree of replication \( r \), \( n \) workers, and a set of queries \( \mathcal{Q} \) on some or all of the snapshots, find a replica distribution \( d = \{V_{i,j,w} : i = 1, 2, \ldots \text{ and } j = 1, 2, \ldots, r \text{ and } w = 1, 2, \ldots, n\} \) that minimizes \( \sum_{q \in \mathcal{Q}} \text{time}(q, d) \) where \( V_{i,j,w} \) denotes the set of vertices that are from the \( j \)th replica \( G_{i,j}(V_{i,j}, E_{i,j}) \) of snapshot \( G_i(V_i, E_i) \) and that are assigned to worker \( w \), and \( \text{time}(q, d) \) denotes the execution time of query \( q \) on the distributed snapshot replicas \( d \) satisfying (1) \( \bigcup_{w=1}^{n} V_{i,j,w} = V_{i,j} = V_i \) for \( j = 1, 2, \ldots, r \) (i.e., the parts of a snapshot replica on all workers cover the original replica), (2) \( V_{i,j,w} \cap V_{i,j,w'} = \emptyset \) if \( w \neq w' \) (i.e., workers are assigned disjoint parts of a snapshot replica), and (3) \( V_{i,j,w} \cap V_{i,j',w} = \emptyset \) if \( j \neq j' \) (i.e., no worker \( w \) contains multiple copies of a vertex and its incident edges) which tolerates \( r - 1 \) simultaneous worker failures.
This problem is identical to Problem 1 if we restrict the degree of replication $r$ to 1. Therefore, like Problem 1, it is no easier than PL-1-3-3SAT. It should be noted that, given a solution to Problem 1, that solution can be applied to the placement of each replica.

Chapter 6 presents our approach to the above problem.
CHAPTER 5
Graph Snapshot Distribution

As mentioned in Section 4.1, G* needs to store each graph snapshot on an appropriate number of workers while balancing the utilization of CPU, local disk, and network resources. In contrast to traditional methods for partitioning a static graph [24, 26], G* must determine the location of each vertex and its edges on the fly in response to a streaming influx of continuously generated graph snapshots from external sources. Our dynamic data distribution approach meets these requirements.

In our dynamic approach, each G* worker partitions its graph data into similarly-sized segments with a certain maximum size (e.g., 10GB) so that it can control its load by migrating some segments to other workers (Sections 5.2 and 5.3). Our approach continuously routes incoming messages for updating vertices and edges to appropriate workers with low latency (Section 5.4.1). When a segment becomes full, G* splits that segment into two that are similar in size while maintaining data locality by keeping data accessed together within the same segment (Section 5.4.2). It does all of this while supporting G*’s graph processing operators (Section 5.4.3). But before G* can tackle the challenge of swapping segments, it first needs to determine the most popular queries so that it can make query-aware segment swapping suggestions.

5.1 Tracking the Top-k Query Patterns

Our segment swapping technique (Sections 5.2 and 5.3) requires knowledge of query patterns. For this reason, we developed an algorithm that maintains the top-\(k\) query patterns given a predefined \(k\). This algorithm represents each query pattern with a collection of graph snapshot identifiers based on the graph snapshots accessed
by the queries constituting that query pattern. It maintains the access frequency of each graph snapshot with small memory overhead using a Count-Min Sketch [66], a sublinear space data structure that can be used for distinct count estimations. It allows various queries in data stream summarization to be approximately answered very quickly. Given a fixed-size integer array and an arbitrary number of items, a Count-Min Sketch can estimate the count of each item with a provable guarantee on the estimation error.

Our algorithm for finding query patterns registers a new query pattern for each distinct query until it obtains $k$ query patterns. It then selects, for each new query, the best matching query pattern from the set of $k$ query patterns and then updates the selected pattern using the new query. Let $G_x$ represents the set of graph snapshots that query $x$ accesses. Then, the matching score for query $x$ and query pattern $q$ is defined as $\frac{\sum_{g \in G_x} P_{R_q}(g)}{|G_x|}$ where $P_{R_q}(g)$ denotes the probability that the queries constituting query pattern $q$ access graph snapshot $g$.

Our technique initially obtains $Q_k$ (equivalently, $k$ popular combinations of segments queried together) in one of two ways.

1. If we are starting cold, initialize the count-min sketch to null and $Q_k$ (for storing $k$ popular query patterns) to $\emptyset$. Then, as queries enter the system, update the count-min sketch for each graph segment that the query accesses with the ID of that query. We estimate the popularity of each combination of segments by consolidating the count-min sketch [67] for those segments as follows: for single-graph queries, we subtract the other count-min sketches; for multi-graph queries, we intersect the count-min sketches of those involved. (See example in Figure 5.2.) Whenever (and while) $|Q_k| > k$, remove its least popular element.

2. If there is an existing collection of graphs and queries, initialize the count-min
sketch to null. Then, for every graph segment, count the number of queries
in which it is referenced (its use-count) and store that in the count-min sketch
indexed on query ID. Sort the graph segments in order of decreasing popularity
by their use-count values. Initialize $Q_k$ (for storing $k$ popular query patterns)
with the first (and therefore most popular) graph segment. Then, for each of
the remaining segments, combine it with each element from $Q_k$, determine the
popularity of this combination of segments as above, and insert the result back
into $Q_k$. Whenever (and while) $|Q_k| > k$, remove its least popular element.

Regardless of how we start, from then on, whenever a query accesses a graph
segment, the associated count-min sketch is updated using the ID of the query.

5.2 Segment Swapping

When a series of graph snapshots is distributed over multiple workers, both the number of snapshots queried and the distribution of the graph snapshots affect query execution time. For example, if most queries access a single graph snapshot, it is advantageous to evenly distribute each snapshot on all workers (i.e., maximize the degree of parallelism). On the other hand, if most queries access all graph snapshots, it is preferable to distribute each snapshot on fewer workers as long as the overall graph data are balanced over all workers. In this way, the system can reduce network overhead while benefiting from high degrees of parallelism.

We developed a technique that distributes each graph snapshot on an appropriate number of workers. In this technique, each worker periodically exchanges graph data with another worker, chosen by a planning heuristic (Section 5.3), in a manner that minimizes the overall expected query execution time.

Given a pair of workers, our technique estimates, for each segment, a heuristic
score representing the benefit of migrating that segment to the other worker, and then performs the most beneficial migration. The benefit/score of migrating a segment is calculated in terms of the expected reduction in query time (i.e., the difference between expected query times before and after migration) or the expected improvement of segment locality.

5.3 Suggesting Good Pairs via Scoring Segment Exchanges

We developed an adaptive segment swap scoring heuristic that enables us to achieve advantageous query-aware segment placement among many servers without exhaustively searching the entire space of possible distributions. It consists of two components: load balancing, described in Section 5.3.1 and locality, described in Section 5.3.2.

In G*, each worker periodically communicates with another worker to balance graph data. That other worker is chosen based on our planner, which suggests pairs via load-balancing or segment locality goals. Our key principle in load balancing is to maximize the benefits of parallelism by uniformly distributing data that are queried together. Our key principle in segment locality is to minimize network overhead by co-locating data from the same snapshot.

5.3.1 Load Balance Goals

Consider Figure 5.1(a) where three snapshots (G_1, G_2, and G_3) have been partitioned into six similarly-sized segments. In this example, workers α and β are assigned a segment from snapshot G_1, α is assigned both segments from G_2, and β is assigned both segments from G_3. If snapshots G_1 and G_2 are frequently queried together (see those shaded in Figure 5.1(a)), this snapshot distribution leads to inefficient query execution due to imbalanced workload between the workers and network communi-
Figure 5.1: Exchanging Segments. If snapshots $G_1$ and $G_2$ are queried together frequently, workers $\alpha$ and $\beta$ in Figure 5.1(a) can better balance the workload and reduce the network overhead by swapping $G_{1,1}$ and $G_{3,1}$.

Given a pair of workers, our technique estimates, for each segment, the benefit of migrating that segment to the other worker, and then performs the most beneficial migration. This process is repeated a maximum number of times or until the migration benefit falls below a predefined threshold. The benefit of migrating a segment is calculated by multiplying the probability that the segment is queried (computed by tracking, in a Count-Min Sketch (Section 5.1), the number of times each query has been executed) with the expected reduction in query time (i.e., the difference between expected query time before and after migration).

For a set $S_i$ of segments on worker $i$ and another set $S_j$ of segments on worker $j$, the expected query time is computed as $\sum_{q \in Q_k} p(q) \cdot time(q, S_i, S_j)$ where $Q_k$ is a collection of $k$ popular query patterns, $p(q)$ is the probability that query pattern $q$ is executed, and $time(q, S_i, S_j)$ denotes the estimated duration of $q$ given segment placement.

While assigning $G_1$ and $G_2$ to different workers prevents $G^*$ from taking advantage of the commonalities between $G_1$ and $G_2$, we assume that query execution time is most affected by network overhead, which has been usually observed in actual deployments of our $G^*$ system.
Algorithm 1: find_partner($S$, $Q$)

input : server $S$, set of queries $Q$
output : the best partner $S'$ for server $S$
1 $S' \leftarrow \text{null}$
2 $\text{max\_benefit} \leftarrow 0$
3 $\text{foreach server } S_i \neq S \text{ do}$
4 \hspace{1em} $(\text{segment\_pair}, \text{benefit}) \leftarrow \text{find\_swap}(S_i, S, Q)$ \hspace{0.5em} // Algorithm 2
5 \hspace{1em} if $\text{benefit} > \text{max\_benefit}$ then
6 \hspace{2em} $\text{max\_benefit} \leftarrow \text{benefit}$
7 \hspace{2em} $S' \leftarrow S_i$
8 return $S'$

ments $S_i$ and $S_j$. We compute $time(q, S_i, S_j)$ as $\max(c(q, S_i), c(q, S_j)) + c'(q, S_i, S_j)$ where $c(q, S_i)$ is the estimated duration of processing the segments from $S_i$ for query $q$, and $c'(q, S_i, S_j)$ represents the estimated time for exchanging messages between workers $i$ and $j$ for query $q$. This is implemented in the algorithms that follow.

Algorithm 1, find_partner, finds, for a given worker/server $S$ and a set of popular queries $Q$, the best partner server $S'$ with which to make a pair $(S, S')$ for segment swapping consideration. It simply calls Algorithm 2, find_swap, pairing every server that’s not $S$ with $S$, and keeps track of the maximum benefit and the server from which that benefit was derived (lines 5–7).

Algorithm 2, find_swap, takes references to two worker/servers, $S_a$ and $S_b$, and a set of popular queries $Q$, in order to return the best pair of segments from those servers to swap along with the cost benefit (in terms of expected query execution time) of that swap. It does this by computing, for all segments $s_i$ from $S_a$ paired with all segments $s_j$ from $S_b$ (lines 3–4), the expected query cost before and after swapping $s_i$ and $s_j$ (lines 5–8) and keeping track of the maximum benefit (lines 10–11) and the segment pair from which that benefit can be derived (line 12). Before and after costs are calculated by Algorithm 3, est_query_times.

Algorithm 3, est_query_times takes a set of popular queries $Q$ and references
Algorithm 2: find_swap($S_a$, $S_b$, $Q$)

```
input : server $S_a$, server $S_b$, set of queries $Q$
output : (best_segment_pair, max_benefit)
1 best_segment_pair ← null
2 max_benefit ← 0
3 foreach segment $s_i$ on server $S_a$ do
4   foreach segment $s_j$ on server $S_b$ do
5     cost_before ← est_query_times($Q$, $S_a$, $S_b$) // Algorithm 3
6     swap $s_i$ and $s_j$ on $S_a$ and $S_b$
7     cost_after ← est_query_times($Q$, $S_a$, $S_b$) // Algorithm 3
8     swap $s_j$ and $s_i$ on $S_a$ and $S_b$ // unswap
9     benefit ← cost_before - cost_after
10    if benefit > max_benefit then
11       max_benefit ← benefit
12       best_segment_pair ← ($s_a$, $s_b$)
13 return (best_segment_pair, max_benefit)
```

Algorithm 3: est_query_times($Q$, $S_a$, $S_b$)

```
input : set of queries $Q$, server $S_a$, server $S_b$
output : estimated time for running queries in $Q$ on servers $S_a$ and $S_b$
1 return $\sum_{q\in Q} Pr(q) \cdot \max\{est_query_time(q, S_a), est_query_time(q, S_b)\}$
   // Algorithm 4
```

to two worker/servers, $S_a$ and $S_b$, in order to return the estimated cost in execution
time of running the queries in $Q$ on servers $S_a$ and $S_b$ given their current segment
placement. This calculation (line 1) is the summation described in Section 5.3.1,
which uses Algorithm 4, est_query_time within it.

Algorithm 4, est_query_time, when given a query $q$ and a worker/server $S$, esti-
mates the cost in time of running query $q$ on server $S$. This calculation (lines 1–3) is
consistent with the assumptions made in the GRASP simulator (Section 7.2), which
are described and explained in Section 7.2.1.
Algorithm 4: est_query_time(q, S)

**input**: query q, server S  
**output**: estimated time for running query q on server S

1. $time_{CPU} \leftarrow |segments(q, S)| \cdot \text{cost}_{CPU}(q)$
2. $time_{Disk} \leftarrow |segments(q, S)| \cdot \text{cost}_{Disk}$
3. $time_{LAN} \leftarrow |external\_edges(q, S)| \cdot \text{cost}_{LAN}(q)$
4. **return** $time_{CPU} + time_{Disk} + time_{LAN}$

### 5.3.2 Segment Locality Goals

If the load among servers is equally (or close to equally) distributed then load balancing heuristics are useless. When this happens, we turn to a different scoring heuristic: the locality of segments on each server.

The segment locality of any given server is defined as the maximum number of segments on that server that are both used in the popular query patterns and are from the same graph, divided by the count of all segments on that server that are part of the popular query patterns. This is computed by Algorithm 5, `segment_locality`, which takes as parameters a worker/server S and a set of popular queries Q. It checks whether or not each segment $s_i$ currently placed on server S (line 3) is part of the set of popular queries Q (line 4). If so, the pattern segment count $ps$ and the value at index $s_i$ in the `graphsUsed` hash are both incremented. Once all all segments have been checked, it returns the maximum count from the `graphsUsed` hash divided by the total pattern segment count $ps$.

We look at segment locality for all workers and return a worker with the lowest segment locality and a worker with the highest segment locality that is still less than 100%. Workers at these extremes in terms of segment locality are good swap pairs because the worker with high locality needs fewer segments to get to 100%, while the worker with low locality contains more distinct segments, and is therefore more likely to have what the other server needs. Workers with 100% locality are taken out of
Algorithm 5: segment locality \((S, Q)\)

\[
\begin{align*}
\text{input} & : \text{server } S, \text{ set of queries } Q \\
\text{output} & : \text{segment locality for } S \\
1 & \quad ps \leftarrow 0 \\
2 & \quad \text{graphsUsed} \leftarrow [] \\
3 & \quad \textbf{foreach} \quad \text{segment } s_i \text{ on server } S \quad \textbf{do} \\
4 & \quad \quad \textbf{if} \quad s_i \text{ is used in } Q \quad \textbf{then} \\
5 & \quad \quad \quad ps++ \\
6 & \quad \quad \quad \text{graphsUsed}[s_i]++ \\
7 & \quad \textbf{return} \quad \text{max(\text{graphsUsed})} / ps
\end{align*}
\]

consideration in future planning.

5.4 Integration with G*

The scoring heuristic is not only quite useful, but it also integrates quite nicely with the existing G* system.

5.4.1 Updates of Vertices and Edges

Each new vertex (or any edge that emanates from the vertex) is first routed to a worker chosen according to the hash value of the vertex ID. That worker assigns such a vertex to one of its data segments while saving the \((\text{vertex ID}, \text{segment ID})\) pair in an index similar to the CGI (Section 3.3). If a worker receives an edge that emanates from an existing vertex \(v\), it assigns that edge to the segment that contains \(v\). If a worker \(w\) has created a segment \(S\) and then migrated it to another worker \(w'\) for load balancing reasons (Section 5.3.1), worker \(w\) forwards the data bound to \(S\) to \(w'\). To support such data forwarding, each worker keeps track of the worker location of each data segment that it has created before. Updates of vertices and edges, including changes in their attribute values, are handled as in the case of edge additions. This assignment of graph data to workers is scalable because it distributes the overhead of managing data over workers. It also proceeds in a parallel, pipelined fashion without
any blocking operations.

5.4.2 Splitting a Full Segment

If the size of a data segment reaches the maximum (e.g., 10GB), the worker that manages the segment creates a new segment and then moves a half of the data from the previous segment to the new segment. To minimize the number of edges that cross segment boundaries, we use a traditional graph partitioning method [24]. Whenever a segment is split as above, the worker also updates the \((vertex \ ID, \ segment \ ID)\) pairs for all of the vertices migrated to the new segment. This update process incurs relatively low overhead since the index can usually be kept in memory as in the case of the CGI (Section 3.3). If a worker splits a segment which was obtained from another worker, it sends the update information to the worker that originally created it in order to enable data forwarding as mentioned in Section 5.4.1.

5.4.3 Supporting Graph Processing Operators

G*’s graph processing operators, such as those for computing clustering coefficients, PageRank, or the shortest distance between vertices, are usually instantiated on every worker that stores relevant graph data [1]. These operators may exchange messages to compute a value for each vertex (e.g., the current shortest distance from a source vertex). If an operator needs to send a message to a vertex, the message is first sent to the worker whose ID corresponds to the hash value of the vertex ID. This worker then forwards the message to the worker that currently stores the vertex. This forwarding mechanism is similar to that for handing updates of vertices and edges (Section 5.4.1).
5.5 Summary

In this chapter we defined how to track the top-$k$ query patterns (for a given $k$) using a Count-Min Sketch. We explained how segment swapping is done by suggesting good worker/server pairs according to our two heuristics (resource balancing and graph segment locality). Finally, we showed that these techniques integrate with our existing G* system. Having solved our query-oriented distribution problem, we now turn our attention to replication.
Time 0: startup
Graphs = [
] Synopses = null
top-k = {}

Time 1: query (id 1) on graph G1
Graphs = [G1] Synopses = G1:x top-1 = {G1}

time 2: query (id 2) on graph G2
Graphs = [G1, G2] Synopses = G1:x-G2:-x top-1 = {G1} or {G2} top-2 = {G1, G2}

Time 3: query (id 3) on graph G1
Graphs = [G1, G2] Synopses = G1:x-G2:-x top-1 = {G1} or {G2} top-2 = {G1, G2}

Time 4: query (id 4) on graph G2
Graphs = [G1, G2] Synopses = G1:x-G2:-x top-1 = {G1} or {G2} top-2 = {G1, G2}

Time 5: query (id 5) on graph G3
Graphs = [G1, G2, G3] Synopses = G1:x-x-x-x-G2:-x-x-x top-1 = {G1} or {G2} top-2 = {G1, G2}

time 6: query (id 6) on graphs G1 and G2 together
Graphs = [G1, G2, G3] Synopses = G1:x-x-x-x-G2:-x-x-x top-1 = {G1} or {G2} top-2 = {G1, G2}

time 7: query (id 7) on graphs G4 and G3 together
Graphs = [G1, G2, G3, G4] Synopses = G1:x-x-x-x-x-x-G2:-x-x-x-x-x top-1 = {G1} or {G2} top-2 = {G1, G2}

Time 8: query (id 8) on graphs G4 and G2 together
Graphs = [G1, G2, G3, G4] Synopses = G1:x-x-x-x-x-x-G2:-x-x-x-x-x top-1 = {G1} or {G2} top-2 = {G1, G2} top-3 = {G1, G2, G3} or {G1, G2, G4} top-4 = {G1, G2, G3, G4}

Time 9: query (id 9) on graphs G1, G2, G3, G4, G5 (all graphs)
Graphs = [G1, G2, G3, G4, G5] Synopses = G1:x-x-x-x-x-x-x-x-G2:-x-x-x-x-x-x-x-x top-1 = {G1} or {G2} top-2 = {G1, G2} top-3 = {G1, G2, G3} or {G1, G2, G4} top-4 = {G1, G2, G3, G4} top-5 = {G1, G2, G3, G4, G5}

Figure 5.2: Maintaining top-k popular graphs and queries
CHAPTER 6
Graph Snapshot Replication

New problems with replication naturally arise in this environment of continuously generated graph snapshots. Although the primary focus of this dissertation is on techniques for query-aware graph series distribution, we developed a replication approach that both accompanies and augments it. This chapter describes our replication approach.

Let’s review what we know so far:

- We know the top-\(k\) popular graphs among all graphs in the system (Section 5.1).
- We know the top-\(k\) popular queries which access those graphs (Section 5.1).
- We know which graph segment placement distributions are beneficial for which queries (Section 5.3).

We also know that workers fail, networks die, and the best laid plans of mice and men often go awry. In order to adapt to this inconvenient reality, and in effort to provide a highly available system, we take up the challenge of replication. (See Section 4.2 for a formal definition of our replication challenge.)

6.1 Replica Creation

We developed a new data replication technique that speeds up queries by configuring replica storage to benefit different categories of queries. This approach masks \(k\) can be configured by the system administrator.
Figure 6.1: Replica Segment Placement where $r = 2$. Both (a) and (b) show the same segment placement for $r = 2$ replication. The system can withstand the loss of any one of the three servers with no loss of data. Category I (single-graph) queries (on graph $A$ in this example) utilize the workers shaded in (a). Category II (all-graph) queries utilize the workers shaded in (b).

up to $r - 1$ simultaneous worker failures by creating $r$ replicas of each graph data segment. It classifies queries into categories and then assigns the replicas of each data segment to a worker in a manner optimized for those categories. For example, assume two query categories which represent queries on a single graph snapshot (Category I) and queries on all snapshots (Category II), respectively. Each graph snapshot replica for Category I could be distributed over all workers to parallelize queries to the maximum extent. On the other hand, each graph snapshot replica for Category II could be distributed over fewer workers to reduce network overhead. (The overall distribution of the graph data still needs to be balanced to effectively parallelize queries.)

As shown in Section 4.2, the optimal distribution of each graph snapshot over workers may vary with the number of snapshots frequently queried together. Based
on this observation, we developed a new data replication technique that speeds up queries by configuring the storage of replicas to benefit different categories of queries. This approach uses an online clustering algorithm [68] to classify queries into $r$ categories based on the number of graphs that they access. It also takes into account the operators that comprise the query and whether or not they require sending and receiving data between vertices. (E.g., the PageRank operator causes much more network overhead than the degree operator.) It then assigns the $j$-th replica of each data segment to a worker in a manner optimized for the $j$-th query category. Continuing our example where $r = 2$, there are two query categories: I and II. We distribute replicas for beneficial placement for representative popular queries as shown in Figure 6.1.
6.2 Query-Aware Replica Selection

When a query is submitted to the G* system, the Query Categorizer component of the master (shown in Figure 6.1) determines into which query category, among the \( r \) categories, the submitted query best fits according to its operators and the number of graphs it touches. It then routes the query to the graph snapshot replicas that are optimized for the chosen query category. For example, queries on a single graph snapshot (Category I) would be executed on the graph snapshot replica placement for Category I (i.e., the replica distributed over all workers, shown in Figure 6.1(a)).

For each query, the Replica Coordinator on the master identifies the worker locations of the data segment replicas to process. To this end, it keeps track of the mapping between graph snapshots and the data segments that constitute them. The master also maintains the mapping between data segment replicas and the workers that store them. Using these mappings, the master selects one replica for each data segment such that the overall processing load is uniformly distributed over a large number of workers and the expected network overhead is low. See Figure 6.2 for details of this example. Finally, just as in Figure 3.7, the master instantiates operators on these workers and starts executing the query.

6.3 Integration with G*

6.3.1 Load Balancing

Each worker balances its graph data as explained in Section 5.3. The only difference regarding replication is that whenever a query of category \( j \) accesses a replica of a data segment, the counting synopsis of the \( j \)-th replica of the data segment is updated using the ID of the query (Section 5.3.1). In this way, the \( j \)-th replica of each segment is assigned to a worker in a manner optimized for query category \( j \).
6.3.2 Updates of Vertices and Edges

Updates of vertices and edges are handled as described in Section 5.4.1 except that each update is routed to $r$ data segment replicas on different workers. For this reason, each worker keeps a mapping that associates each segment ID with the $r$ workers that store a replica of the segment. Our approach protects this mapping on worker $w$ by replicating it on workers $(w+1) \mod n$, $(w+2) \mod n$, $\ldots$, $(w+r-1) \mod n$ where $n$ denotes the number of workers. If a worker fails, the master assigns another worker to take over.

6.3.3 Splitting a Full Segment

The replicas of a data segment are split in the same way due to the use of a deterministic partition method. For each vertex migrated from one data segment to another, the $r$ workers that keep track of that vertex update their ($vertex ID$, $segment ID$) pairs accordingly.

6.3.4 Supporting Graph Processing Operators

As noted in Section 5.4.3, G*'s graph processing operators are usually instantiated on every worker that stores relevant graph data. The difference here is that the Replica Coordinator (shown in Figure 6.2) must tell the Query Coordinator (shown in Figure 3.1) on what workers to instantiate the query processing operators. These operators may exchange messages to compute a value for each vertex (e.g., the current shortest distance from a source vertex). If an operator needs to send a message to a vertex, the message is first sent to the worker whose ID corresponds to the hash value of the vertex ID. This worker then forwards the message to the worker that currently stores the vertex. This forwarding mechanism is similar to that for handing updates of vertices and edges (Section 5.4.1).
6.4 Summary

In this chapter we presented our approach for graph series replica creation and query-aware replica selection for advantageous query execution. We also showed that these techniques integrate with our existing G* system.
CHAPTER 7
Evaluation Results

In order to evaluate our graph segment distribution techniques described in Chapter 5 we experimented in two environments. The first environment was our 48-core server cluster. The second was our simulator, which we built in order to explore how our segment distribution techniques are affected by the various parameters we can adjust.

7.1 Server Cluster Results

This section presents our results obtained by running G* on a six-node, 48-core server cluster. In this cluster, each machine has two Quad-Core Xeon E5430 2.67 GHz CPUs, 16GB RAM, and a 2TB hard drive.

To construct the realistic example used in Section 4.1, we measured the overhead of key operations. In our evaluation, a worker was able to transmit up to 1 million messages to other workers within a second, although a 1Gb/second connection may enable 10 million transmissions of 12-byte messages in theory. The reason behind this result is that there is inherent overhead when writing and creating message objects to and from TCP sockets. Furthermore, reading approximately 1GB of data from disk to construct a graph snapshot took 5 seconds. However, constructing a snapshot in memory by creating 100 million edge objects and registering them in an internal data structure took approximately 200 seconds.

<table>
<thead>
<tr>
<th># Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.0x</td>
<td>1.9x</td>
<td>3.7x</td>
<td>5.9x</td>
<td>9.7x</td>
<td>12.5x</td>
<td>14.7x</td>
</tr>
</tbody>
</table>

Table 7.1: Preliminary Actual Speedup Results
In the next set of experiments, we created a series of 500 graph snapshots using a binary tree generator. Each snapshot in the series was constructed by first cloning the previous snapshot and then inserting 20,000 additional vertices and edges to the new graph. Therefore, the last graph in the series contained 10 million vertices. We ran a query that computes, for each graph, the distribution of the shortest distances from the root to all other vertices. Table 7.1 shows, for the shortest distance query, the speedup achieved by distributing the snapshots over more cores in more workers. The highest speedup was achieved with 48 cores. This table also shows that the relative benefit of data distribution (i.e., the speedup relative to the number of workers) tends to decrease with more workers. This is mainly due to increased network traffic, which shows the importance of balancing CPU and network resources in the context of continuously created large graph snapshots.

The effectiveness of two different distributions is demonstrated in Table 7.2. If most queries access only the largest snapshot, then it is beneficial to distribute that snapshot over all workers to maximize query speed. On the other hand, if all of the snapshots are queried together, our approach stores each graph on a smaller subset of workers to reduce network overhead. In this case, all of the workers can still be used in parallel since the entire graph data is distributed over all workers. The benefits of distribution configurations are less pronounced in Table 7.2 than Table 4.1 due to a smaller number of message transmissions and fewer workers (because one is a PageRank query – which is many iterations (e.g., 10 in this case) – and the other is a shortest path query). Table 7.2 also demonstrates the benefit of G* executing queries

<table>
<thead>
<tr>
<th></th>
<th>SSSP Query</th>
<th>All Workers</th>
<th>Subset of Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>One snapshot</td>
<td>8.2 seconds</td>
<td>19.2 seconds</td>
<td></td>
</tr>
<tr>
<td>All snapshots</td>
<td>80.5 seconds</td>
<td>53.2 seconds</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Preliminary Impact of Graph Data Distribution
on multiple snapshots. In particular, the time for processing 500 snapshots (e.g., 80.5 seconds) is only up to 10 times longer than that for processing the largest snapshot (e.g., 8.2 seconds) since the computations on the largest snapshot are shared.

7.2 The GRASP Simulator

We built the Graph Replication And Smart Partitioning simulator, shown in Figures 7.1 and 7.2, so that we could experiment with the many aspects of graph segment distribution in a variety of scenarios. We used our server cluster evaluation results (Section 7.1) and our cost function (Section 5.2) to guide the building of the simulator.

By turning the assorted “knobs”, we could set and test the following:

- our key assumptions about CPU, Disk, and LAN costs (Table 7.3);
- the effect of the number of worker servers, graphs, and segments per graph;
- initial and model segment distribution conditions;
- the characteristics of popular queries;
- the impact of various pairing heuristics (planned, load-balanced, or random);
the stopping conditions, either as a ratio of the number of swaps made to the number of passes made or as a limit on the number of pairs considered.

For the scenarios we constructed, the simulator measured the following quantities:

- number of pairs considered;
- number of passes made;
- number of swaps made;
- ratio of swaps to passes;
- starting, ending, and current (in real-time) planning heuristic score;
- improvement in terms of heuristic score and speedup factor;
- improvement as a percentage of the selected model distribution.

Given configurable weights for CPU, disk, and network overhead; plus the number of the workers, graphs, and segments per graph; along with the popular queries and starting segment distribution (all set forth in the GRASP GUI), the simulation uses our heuristic planning function explained in Section 5.3 to suggest good servers for which to investigate swaps. This segment swapping process is repeated a maximum number of times, as specified by the stopping condition in the GUI. The simulation computes the time for a query \( q \) on the suggested pair of servers, \( S_i \) and \( S_j \), as \( \max(c(q, S_i), c(q, S_j)) + c'(q, S_i, S_j) \) where \( c(q, S_i) \) is the estimated duration of processing the segments from \( S_i \) for query \( q \), and \( c'(q, S_i, S_j) \) represents the estimated time for exchanging messages between workers \( i \) and \( j \) for query \( q \).

In order to estimate the duration of processing segments and the cost of exchanging messages between workers on the network we had to make some key assumptions.
Figure 7.2: The GRASP simulator: Results
7.2.1 Key Assumptions

Our approach requires that each G* worker partition its graph data into similarly-sized segments of a certain maximum size. Fixing the graph segment size at 10GB, we made the assumptions noted in Table 7.3. Justification of those assumptions follows.

*Cost*$_{CPU}$: This includes RAM read, write, and access time as well as computational overhead, operating system overhead (including context switches, TLB misses, and more) in addition to object serialization and de-serialization.

*Cost*$_{Disk}$: Assuming we can read at a sustained rate of 200MB/second, it will take 50 seconds to read each 10GB graph segment into RAM. The recent SATA III specification maxes read speed at 600MB/second, but as a practical matter no current hard disk drive can keep up with that for long periods, even for sequential reads. There are buffer issues, congestion issues, and all kinds of overhead. We take more realistic, reasonable, and Google-supported [69] approach and use 200MB/second, while still ignoring some finer details including but not limited to bus type, rotational speed, number of platters, capacity, density, and a host of firmware and protocol issues.

*Cost*$_{LAN}$: We assume messages about one 10GB graph segment are themselves roughly 10GB in size because $sizeOf\{\text{source, dest, weight}\} \approx sizeOf\{\text{source, dest, aggregate value}\}$. We further assume we have at our disposal 10Gb/second switched Ethernet, which equates to roughly 1.25GB/second, and less roughly about 1.0GB/second sustained throughput when considering overhead including parity bits and other error detection and correction. This means it takes about 10 seconds to exchange 10GB of data among all servers in parallel. There is also packet overhead.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Cost</em>$_{CPU}$</td>
<td>0.1</td>
<td>seconds to access RAM and perform computations for one segment</td>
</tr>
<tr>
<td><em>Cost</em>$_{Disk}$</td>
<td>50</td>
<td>seconds to read one segment from disk into RAM</td>
</tr>
<tr>
<td><em>Cost</em>$_{LAN}$</td>
<td>10</td>
<td>seconds to send messages for one segment over the network</td>
</tr>
</tbody>
</table>

Table 7.3: Key Simulation Assumptions
ACKnowledgements, packet-to-frame and frame-to-packet delay, and several of other aspects which we only estimate in our model.

Each worker/server operates in parallel, so simulator additively accumulates $\text{Cost}_{\text{CPU}}$ and $\text{Cost}_{\text{Disk}}$ for each server and then takes the maximum of those values for the servers involved. The $\text{Cost}_{\text{LAN}}$ estimation is a little more subtle. As such, we discuss it in the next two subsections.

7.2.1.1 Network Parallelism

Many recent Ethernet switches such as Cisco’s Nexus 5000 [70] utilize a crossbar fabric for packet transfer. See Figure 7.5 for details. While this provides full-duplex, point-to-point communications, multiple conversations involving the same source or destination must be interleaved, so their combined time is additive. Newer and more advanced Ethernet switches such as Cisco’s Nexus 7000 [71] utilize parallel fabric modules and can thus achieve full-duplex, point-to-point communications in parallel, so their combined time is the maximum of all the times involved in the parallel communications. See Figure 7.6 in for details.
7.2.1.2 Segment Swapping and Network Cost: An Example

Consider a scenario where we have four workers to manage four graphs \((G_1, G_2, G_3, \text{ and } G_4)\) of 2997 edges each, and every graph is partitioned into three similarly-sized segments of 999 edges by random or hash-based partitioning. Assume that we wish to distribute the segments amongst the servers to best service one query on graph \(G_3\). Assume also that we are operating under the assumptions noted in Table 7.3.

As can been seen from in Figure 7.3, moving graph segment \(G_{3,1}\) from server \(s_3\) to server \(s_0\) results in \(333 \cdot 4 = 1332\) crossing edges, which seems to require 13.3 seconds at half duplex speed or 6.7 seconds at full-duplex speed. The veracity of this assumption, however, depends on the network infrastructure, specifically the switch.

Figure 7.5 depicts a simple model of the switching fabric in a standard Ethernet switch such as Cisco’s Nexus 5000 [70]. With this switch, point-to-point communications that share a common source or destination must share the crossbar paths in an interleaved manner. This results in our needing to accumulate the communication time rather than consider the maximum.

Figure 7.6 depicts a simple model of the switching fabric in an advanced Ethernet switch such as Cisco’s Nexus 7000 [71]. With this switch, point-to-point communications that share a common source or destination can operate in parallel thanks to multiple, parallel crossbar fabrics. In these circumstances, rather than adding all of the communication times as with a standard Ethernet switch, we can take the maximum communication time instead. This is the situation we model in our simulator.
7.3 Simulation Results

Once our simulator was complete, we conducted experiments to measure the impact of graph segment distribution in a variety of environments by increasing the number of workers, graphs, and segments per graph. We kept the number of workers, graphs, and segments per graph the same so that we would be taking maximum advantage of parallelism and testing the efficacy of our planning heuristic in environments growing from $2^8$ to $2^{16}$ total segments. It was also important that we show the efficacy of graph segment placement in parallel computing environments. Having fewer graphs than we have workers, or having fewer segments per graph than we have workers results in less-than-maximal parallelism. (If we had, say 64 servers and 64 graphs, but each graph was broken into fewer than 64 segments, then we would be unable to achieve a fully parallel segment distribution because some of the workers
would be idle in a shared-everything configuration.)

7.3.1 Experiment Details

We first looked at 16 workers managing 16 graphs of 16 segments each. (The “16-16-16” configuration). We then experimented on 32-32-32, 64-64-64, 128-128-128, and 256-256-256 configurations. We tested three methods for suggesting server pairs – planning (as explained in Section 5.3), load balanced, and random – with two query types: a shortest path type query and a ten iteration PageRank type query. To simulate PageRank queries we increased our \( Cost_{CPU} \) and \( Cost_{LAN} \) simulation parameters by a factor of 10 to account for the computation and network overhead of 10 iterations. We ran queries on one graph and on all graphs for each combination of heuristics and query types. For some combinations we also ran queries on first and last graphs and also on every \( x \) graphs. These reasoning behind these analytic use
The figures that follow in this section plot the number of pairs considered on the X-axis against the estimated query time on the Y-axis. The estimated query time was computed as specified in Section 5.2. In each case, the points where X = 0 represent the estimated query time given the initial graph segment distribution (placement) while the solid black line denotes the estimated query time of the model graph segment distribution selected for that experiment. As noted in the legend on each plot, the solid red line represents planned server pairing, the long-dashed green line represents load-balanced server pairing, and the small-dashed blue line represents random server pairing. In many cases, especially where we query a single graph, planned and load-balanced pairing perform similarly, and as such, the red and green lines overlap.

7.3.2 Results

7.3.2.1 16 workers, 16 graphs of 16 segments each

Our initial experimental workload consisted of 16 workers and 16 graphs of 16 segments each, meaning we had a total of 256 segments for which to find beneficial distributions based on the query type and the number of graphs being queried. Figure 7.7 shows our results from simulating the execution of a single iteration shortest path type query on one graph in the 16-16-16 configuration (16 graphs of 16 segments each, with 16 workers over which to distribute the work) that starts out with shared-
nothing replica placement. The initial estimated query time is 801.6 seconds. Our planned pairing and load-balanced pairing both perform very well here, with both finding the model segment placement after considering only 3840 pairs, achieving a 14.63x speedup to only 54.7 seconds. Random pairing, on the other hand, had to consider 98,560 pairs before it found a segment placement equivalent to the model.

Our planning and load balancing heuristics for pair suggestions perform similarly in this and many other – but not all – scenarios. This is because load balancing is one of the aspects considered in our planning heuristic (Section 5.3.1), which suggests a server pair by taking one of the least-loaded servers and finding its best match via Algorithm 1.

Figure 7.8 shows our results from simulating the execution of a multi-iteration PageRank type query on one graph in the 16-16-16 configuration. The results look very similar to those in Figure 7.7. This is to be expected, since the server and graph configuration are the same but the $Cost_{CPU}$ and $Cost_{LAN}$ parameters have been increased by factor of 10 to account for 10-iteration PageRank. The initial estimated
query time is now 816 seconds. Planned and load balanced pairing find the optimal segment placement after considering only 3840 pairs, achieving a 8.34x speedup after only 97.9 seconds.

Changing to queries on the first and last graphs, Figures 7.9 and 7.10 show our results for shortest path and PageRank type queries respectively. Once again planning and load balancing perform equally well, but in interestingly different manners. While load balanced pair suggestion showed the same pattern of behavior as seen earlier in Figures 7.7 and 7.8, our planned heuristic is different. In this scenario, planning actually gets worse before it gets better, but once it gets better, it converges to the model placement at a faster rate than load balancing. (Random pairing still takes a long time.) This is because our planner only looks at the two servers it is considering at that time as it evaluates the possibilities for good pairs. In this manner it avoids being caught in local minima or maxima, a common problem in these sorts of heuristics. But the trade-off is that, in the beginning, the planner will make some swaps that do not seem, at that moment, to be beneficial overall, but that are indeed
Figure 7.9: Shortest Path-type query on first and last graphs (16 servers) beneficial later on, as the plots show.

Moving on to queries on every 4th graph, Figures 7.11 and 7.12 show our results for shortest path and PageRank type queries respectively. As was the case in Figures 7.9 and 7.10 for queries on first and last graphs, planning and load balancing perform equally well, but in interestingly different manners, for the same interesting reasons. As we get closer to querying all of the graphs, the benefit of realigning the initial graph segment placement from shared-nothing to shared-everything gets smaller. We see that here, as this realignment results in speedup of only 3.66x and 2.08x for shortest path and PageRank type queries respectively.

Continuing the trend of querying more graphs, Figures 7.13 and 7.14 show our results for shortest path and PageRank type queries on all graphs, respectively. This time the initial segment placement was shared-everything (which was our goal in the prior cases so far) with the model placement being shared-nothing. When querying only a single graph in this scenario (Figure 7.13) the speedup is a negligible 1.09x, reducing the estimated query time from 876.6 seconds to 801.6 seconds. The segment
placement does not have a significant impact on single iteration queries on all graphs. Looking at Figure 7.14, however, we can see a significant impact on multiple iteration queries on all graphs, as we observe a 1.92x speed up as the estimated time is reduced from 1566 seconds to 816 seconds. Our planner performs well in this scenario.

It is interesting to note, in these cases when we are querying all graphs, that load balance based pairing and random pairing are essentially useless. Load balancing is of no use because we are querying all graphs, so all workers are involved in the query, and thus there are no workers with less load. Random pairing is of no use because every time a beneficial swap is made, the likelihood of randomly finding another one decreases.

**Figure 7.10:** PageRank-type query on first and last graphs (16 servers)
Figure 7.11: Shortest Path-type query on every 4th graph (16 servers)

Figure 7.12: PageRank-type query on every 4th graph (16 servers)
Figure 7.13: Shortest Path-type query on all graphs (16 servers)

Figure 7.14: PageRank-type query on all graphs (16 servers)
7.3.2.2 32 workers, 32 graphs of 32 segments each

Our second experimental workload consisted of 32 workers and 32 graphs of 32 segments each, meaning we had a total of 1024 segments for which to find beneficial distributions based on the query type and the number of graphs being queried.

Shortest path and PageRank queries on this workload exhibit similar characteristics as on our 16-16-16 workload, which shows that our solution scales nicely as we increase the workload. One difference, notable in the queries on every 8th graph, is that our planning heuristic takes quite a bit longer, as a portion of total time, before it “turns the corner” and recognizes globally beneficial swaps.

The similarities continue regarding queries on all graphs. As before, the initial segment placement was shared-everything, with the model placement being shared-nothing. Also as before, the speed up on PageRank, it being 10 iterations, is more pronounced than the speed up we see on shortest-path type queries.
Figure 7.16: PageRank-type query on one graph (32 servers)

Figure 7.17: Shortest Path-type query on every 8\textsuperscript{th} graph (32 servers)
**Figure 7.18:** PageRank-type query on every 8th graph (32 servers)

**Figure 7.19:** Shortest Path-type query on all graphs (32 servers)
Figure 7.20: PageRank-type query on all graphs (32 servers)
7.3.2.3 64 workers, 64 graphs of 64 segments each

Our third experimental workload consisted of 64 workers and 64 graphs of 64 segments each, meaning we had a total of 4096 segments for which to find beneficial distributions based on the query type and the number of graphs being queried.

The scenarios shown here exhibit similar characteristics to their 16-16-16 and 32-32-32 versions, showing that our planning heuristic continues to scale with workload. It is of particular note that at this scale, random pairing is essentially useless. There are just too many possibilities to rely on happenstance.
Figure 7.22: PageRank-type query on one graph (64 servers)

Figure 7.23: Shortest Path-type query on all graphs (64 servers)
Figure 7.24: PageRank-type query on all graphs (64 servers)
7.3.2.4 128 workers, 128 graphs of 128 segments each

Our penultimate experimental workload consisted of 128 workers and 128 graphs of 128 segments each, meaning we had a total of 16,384 segments for which to find beneficial distributions based on the query type and the number of graphs being queried.

For a PageRank-type query on all graphs we looked only at the performance of our planner since both load balance-based worker pairing and random worker pairing have been shown to be of no use in queries on all graphs in our prior scenarios. Once again we see our planner performing very well.
Figure 7.26: PageRank-type query on all graphs (128 servers)
7.3.2.5 256 workers, 256 graphs of 256 segments each

Our final experimental workload consisted of 256 workers and 256 graphs of 256 segments each, meaning we had a total of 65,536 segments for which to find beneficial distributions based on the query type and the number of graphs being queried.

Once again, for a PageRank-type query on all graphs we looked only at the performance of our planner since both load balance-based worker pairing and random worker pairing have been shown to be of no use in queries on all graphs in our prior scenarios. Our planner remains effective, even at this scale.

7.3.3 Summary

Table 7.5 summarizes our results. With the exception of shortest path type queries on all graphs, every other query on every other workload on which we experimented resulted in meaningful speedup. As is evident in the table, segment distribution has the most impact on queries involving fewer graphs, and decreases as more and more graphs are touched by the queries being run. Still, even on queries
involving all graphs, a beneficial distribution can almost double the query speed.

<table>
<thead>
<tr>
<th>Query</th>
<th>Initial Distribution</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-16-16 (256 graph segments)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One graph: Shortest Path</td>
<td>shared-nothing</td>
<td>14.63x</td>
</tr>
<tr>
<td>One graph: PageRank</td>
<td>shared-nothing</td>
<td>8.34x</td>
</tr>
<tr>
<td>First and last graphs: Shortest Path</td>
<td>shared-nothing</td>
<td>7.32x</td>
</tr>
<tr>
<td>First and last graphs: PageRank</td>
<td>shared-nothing</td>
<td>4.17x</td>
</tr>
<tr>
<td>Every 4&lt;sup&gt;th&lt;/sup&gt; graph: Shortest Path</td>
<td>shared-nothing</td>
<td>3.66x</td>
</tr>
<tr>
<td>Every 4&lt;sup&gt;th&lt;/sup&gt; graph: PageRank</td>
<td>shared-nothing</td>
<td>2.08x</td>
</tr>
<tr>
<td>All graphs: Shortest Path</td>
<td>shared-everything</td>
<td>1.09x</td>
</tr>
<tr>
<td>All graphs: PageRank</td>
<td>shared-everything</td>
<td>1.92x</td>
</tr>
<tr>
<td>32-32-32 (1024 graph segments)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One graph: Shortest Path</td>
<td>shared-nothing</td>
<td>29.18x</td>
</tr>
<tr>
<td>One graph: PageRank</td>
<td>shared-nothing</td>
<td>16.41x</td>
</tr>
<tr>
<td>Every 8&lt;sup&gt;th&lt;/sup&gt; graph: Shortest Path</td>
<td>shared-nothing</td>
<td>7.29x</td>
</tr>
<tr>
<td>Every 8&lt;sup&gt;th&lt;/sup&gt; graph: PageRank</td>
<td>shared-nothing</td>
<td>4.10x</td>
</tr>
<tr>
<td>All graphs: Shortest Path</td>
<td>shared-everything</td>
<td>1.10x</td>
</tr>
<tr>
<td>All graphs: PageRank</td>
<td>shared-everything</td>
<td>1.95x</td>
</tr>
<tr>
<td>64-64-64 (4096 graph segments)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One graph: Shortest Path</td>
<td>shared-nothing</td>
<td>58.28x</td>
</tr>
<tr>
<td>One graph: PageRank</td>
<td>shared-nothing</td>
<td>32.57x</td>
</tr>
<tr>
<td>All graphs: Shortest Path</td>
<td>shared-everything</td>
<td>1.10x</td>
</tr>
<tr>
<td>All graphs: PageRank</td>
<td>shared-everything</td>
<td>1.97x</td>
</tr>
<tr>
<td>128-128-128 (16,384 graph segments)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One graph: Shortest Path</td>
<td>shared-nothing</td>
<td>116.47x</td>
</tr>
<tr>
<td>All graphs: PageRank</td>
<td>shared-everything</td>
<td>1.97x</td>
</tr>
<tr>
<td>256-256-256 (65,536 graph segments)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All graphs: PageRank</td>
<td>shared-everything</td>
<td>1.98x</td>
</tr>
</tbody>
</table>

Table 7.5: Simulation Scenarios and Speedup Results
CHAPTER 8
Conclusions and Future Directions

The classic challenges of data distribution and replication are imbued with renewed significance given an environment of continuously generated graph snapshots. In this dissertation, we presented techniques that address these new graph distribution and replication challenges. While our distribution techniques and replication approach effectively solve the problems we presented, rich opportunities for future work remain.

8.1 Conclusions

Our dynamic data distribution approach efficiently stores graph data on the fly using multiple worker servers in parallel. In our approach, each G* worker partitions its graph data into similarly-sized segments with a certain maximum size (e.g., 10GB) so that that worker can control its load by migrating some segments to other workers (Section 5.3.1). Our approach continuously routes incoming messages for updating vertices and edges to appropriate workers with low latency (Section 5.4.1). When a segment becomes full, G* splits that segment into two that are similar in size while maintaining data locality by keeping data accessed together within the same segment (Section 5.4.2). It does all of the above while supporting G*’s graph processing operators (Section 5.4.3). This technique also gradually adjusts the number of workers that store each graph snapshot while balancing network and CPU overhead to maximize overall performance.

Our data replication technique (Chapter 6) maintains each graph replica on a different number of workers, making available the most efficient storage configurations for various combinations of queries. We showed how to integrate these techniques with
G*, our scalable and robust system for storing and querying large graph snapshots.

The efficacy of these techniques was demonstrated in Chapter 7.

8.2 Future Directions

Our study of graph distribution, replication, and query performance in the context of continuously evolving graph snapshots opens several research directions. We intend to consider the following challenges in the future:

- **Additional Experiments Using Real Data Sets.** Our current research considers real-world analytic use-cases (Table 7.4) using simulated and synthetic graph data. Applying our real-world scenarios to larger and more general data sets such as those from Twitter [61] and Yahoo! [72] may prove interesting.

- **Clustering Popular Queries.** We currently track the top- \( k \) popular individual queries (Section 5.1) in order to inform our segment swapping technique (Sections 5.2 and 5.3). Considering a clustering approach to categorizing queries may increase query coverage at the expense of accuracy. We intend to explore this tradeoff.

- **Dynamic Network Environments.** In this dissertation we assumed that the computing environment in which we are running is always available. We further assumed in our model of the computation environment (Section 7.2.1) that this environment is stable. However, availability, speed, capacity, and load of computers and their underlying network infrastructure can change drastically (and sometimes quickly) in public and private cloud computing environments. Exploring real-time network topology-aware distribution and replication is therefore another promising area for further research.
• **New Network Environments.** Even available and stable near-future networking environments will grow to differ from our current assumptions. Two specific areas of immediate interest are as follows:

  – Software defined networking – where traffic flow decisions (in the control plane) are made in software as opposed to being tightly coupled with traffic forwarding hardware (in the data plane) – will fundamentally change the nature of enterprise networks.

  – Data center TCP – Modification of the TCP stack on datacenter nodes that implements Explicit Congestion Notification. This allows the devices in the path to signal congestion and change TCP window sizes on the fly to avoid dropped packets.

These and other advances in routing, switching, virtualized network environments, and dynamic infrastructure present new opportunities for network-aware data distribution, replication, and query execution.

• **Multiple Swaps per Pass.** The approach presented in this dissertation pairs servers by determining the best match for a given server (Algorithm 1). This is done by finding the most beneficial swap of individual segments, one from each server in the pair (Section 5.3). A more general version of swapping that considers multiple swaps per pass may be beneficial. This is challenging because each swap creates a new version of the segment placement “world” so we must account for the cascading effect of multiple swaps in a single pass. Whether or not this will lead to results more promising than what we have already achieved is entirely unclear, but it’s an interesting direction to pursue.

• **Graph Indexes on Relational Databases.** Studying the applicability of using a graph series database such as G* as an additional index on existing
relational database systems is an exciting prospect, both theoretically and commercially.

- **Extending to the Relational Model.** Perhaps most exciting is the prospect of applying all that we have learned about efficiently storing, indexing, querying, optimizing, and making reliable graph series databases (based on versioning with snapshots) to existing and new database systems based on the relational model.
BIBLIOGRAPHY


[16] Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N.: Time-Varying
Graphs and Dynamic Networks. In: Proceedings of the 10th International Con-
ference on Ad-hoc, Mobile, and Wireless Networks (ADHOC-NOW). (2011) 346–
359

Frequent Subgraphs. In: Proceedings of the IEEE Symposium on Computational
Intelligence and Data Mining (CIDM). (2007) 35–42

Networks. CoRR abs/1101.5913 (2011)

Social Network Analysis. In: Proceedings of the 2nd ACM Workshop on Online

SIGKDD International Conference on Knowledge Discovery and Data Mining

[21] Neely, M.J., Modiano, E., Rohrs, C.E.: Dynamic Power Allocation and Rout-
ing for Time Varying Wireless Networks. In: Proceedings of the 22nd Annual
Joint Conference of the IEEE Computer and Communications IEEE Societies

[22] DeWitt, D., Gerber, R., Graefe, G., Heytens, M., Kumar, K., Muralikrishna,
M.: Gamma-a high performance dataflow database machine. In: Proceedings of
the 12th International Conference on Very Large Data Bases. (1986) 228–237

[23] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large


[34] Neo4j. http://neo4j.org/


[61] Twitter Streaming API. https://dev.twitter.com/docs/streaming-api/methods


