A Demonstration of Query-Oriented Distribution and Replication Techniques for Dynamic Graph Data*

Alan G. Labouseur, Paul W. Olsen Jr., Kyuseo Park, and Jeong-Hyon Hwang
University at Albany – State University of New York
Albany, New York 12222 USA
{alan, polsen, kpark, jhh}@cs.albany.edu

ABSTRACT

Evolving networks can be modeled as series of graphs that represent those networks at different points in time. 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. If multiple graph snapshots are commonly queried together, traditional techniques that distribute data over all servers or create identical data replicas result in inefficient query execution.

We propose to verify, using live demonstrations, the benefits of our graph snapshot distribution and replication techniques. Our distribution technique adjusts the set of servers storing each graph snapshot in a manner optimized for popular queries. Our replication technique maintains each snapshot replica on a different number of servers, making available the most efficient replica configurations for different types of queries.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel Databases*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Databases*

Keywords

load balancing; replication; graph partitioning; parallel processing; high availability

1. INTRODUCTION

We are surrounded by constantly evolving networks, including social networks, citation networks, transportation networks, and the Web [1]. We can take periodic snapshots of these networks and model them as graphs where vertices represent entities and edges represent relationships between entities. Trends discovered by analyzing the evolution of a network via these *graph snapshots* play a crucial role in many areas, such as sociopolitical science, viral marketing, and national security. G* is our distributed system for managing series of large graph snapshots representing an evolving network at different points in time [5, 7, 11]. It efficiently stores and queries graph snapshots on multiple servers by taking advantage of commonalities among them.

Accelerating computation by distributing data over multiple servers has been a popular approach in parallel databases [4] and distributed systems [3]. Techniques for partitioning individual graphs to facilitate parallel computation have also been developed [6, 8, 9, 10]. 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*. We require solutions that (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 those snapshots. To the best of our knowledge, G* is the first system to address these concerns.

We propose to demonstrate our technique that tackles the above challenges by splitting graphs into similarly sized segments and then swapping segments between pairs of servers such that the overall expected query time is reduced [7]. To calculate the expected query time, our technique categorizes previously submitted queries while keeping track of the CPU and network overhead as well as the frequency of each query category. We also propose to present our technique that tolerates up to $r - 1$ simultaneous server failures by constructing r replicas for each graph snapshot. To improve query performance, this technique classifies queries into r categories and optimizes the distribution of each replica for one of the query categories.

Our demonstration will use both real-world and synthetic data sets to allow attendees to experience for themselves the following aspects:

*This work is supported by NSF CAREER Award IIS-1149372.

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2577026.

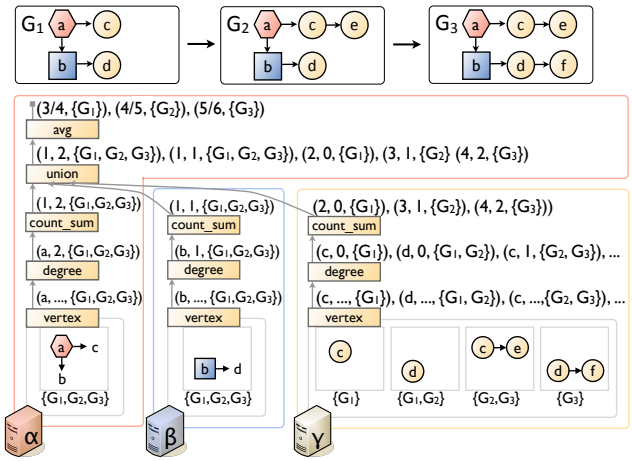


Figure 1: Deduplicated Storage and Querying of Graph Snapshots (Average Degree per Snapshot)

- the impact of graph snapshot distribution
- the benefits of segment swapping
- the effectiveness of query categorization
- the advantages of graph snapshot replication

The remainder of this demonstration proposal presents our new graph snapshot distribution and replication techniques (Section 2) as well as specific demonstration scenarios emphasizing the impact and effectiveness of these techniques (Section 3).

2. G* BACKGROUND

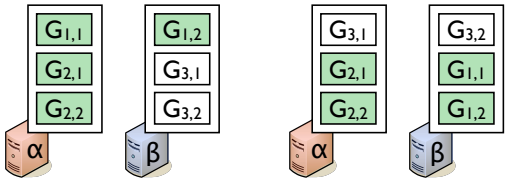
This section describes the architecture of the G* system (Section 2.1) and our techniques for distributing graph snapshots (Sections 2.2, 2.3, and 2.4) as well as replicating them (Sections 2.5 and 2.6).

2.1 Architecture

G* consists of a *master* and multiple *workers*. As Figure 1 shows, graph snapshots (e.g., G_1, G_2, and G_3) are distributed over workers (e.g., α, β, and γ). In response to each submitted query, the master (not shown in Figure 1) constructs a network of operators that process data on workers in parallel. G* supports deduplicated storage and querying of graph snapshots. For example, in Figure 1, vertex a and its edges remain the same in graph snapshots G_1, G_2, and G_3. Therefore, they are stored only once on a worker. Given a query for finding the average degree on each of G_1, G_2, and G_3, the degree of a is also computed only once. Further details of our storage and query techniques can be found in our previous publications [5, 11]. G* is currently implemented in approximately 30,000 lines of Java code (`http://www.cs.albany.edu/~gstar/`).

2.2 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.



(a) Before Exchange (b) After Exchange

Figure 2: Exchanging Segments. If snapshots G_1 and G_2 are queried together frequently, workers α and β in Figure 2(a) can better balance the workload and reduce the network overhead by swapping $G_{1,1}$ and $G_{3,1}$.

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 (randomly chosen) worker in a manner that minimizes the overall expected query execution time. Consider Figure 2(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 2(a)), this snapshot distribution leads to inefficient query execution due to imbalanced workload between the workers and network communications for the edges between $G_{1,1}$ and $G_{1,2}$. Exchanging $G_{1,1}$ and $G_{3,1}$ between the workers solves this problem[1]. In this case, the frequently queried data (i.e., G_1 and G_2) are balanced over the workers. Each of G_1 and G_2 can also be processed only on workers β and α, respectively, without incurring high network overhead.

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. The benefit 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). Given a set S_i of segments on worker i and another set S_j of segments on worker j, the expected query time is defined as $\sum_{q \in \mathcal{Q}_k} \Pr(q) \cdot time(q, S_i, S_j)$ where $\mathcal{Q}_k$ is a collection of k popular query patterns, $\Pr(q)$ is the probability that query pattern q is executed, and $time(q, S_i, S_j)$ denotes the estimated duration (based on configurable weights for CPU and network overhead observed in our G* system) of q given segment placements S_i and S_j. The segment swapping process is repeated a maximum number of times or until the migration benefit falls below a predefined threshold.

2.3 Finding Query Patterns

Our segment swapping technique (Section 2.2) requires knowledge of query patterns. For this reason, we developed an algorithm that maintains k query patterns given

[1]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.

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 also maintains the access frequency of each graph snapshot with small memory overhead using a *count-min sketch* [2]. 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 $\frac{\sum_{g \in G_x} \Pr_q(g)}{|G_x|}$ where $\Pr_q(g)$ denotes the probability that the queries constituting query pattern q access graph snapshot g.

2.4 Segment Splitting

When a graph segment reaches a predefined maximum size, the worker responsible for that segment creates a new segment and moves half of the data there. For this, we use a traditional graph partitioning method [6] to minimize the number of edges that cross segment boundaries.

2.5 Graph Snapshot Replication

G* masks up to $r-1$ simultaneous worker failures by creating r replicas of each graph data segment. We developed a new data replication technique that speeds up queries by configuring the storage of these replicas to benefit different categories of queries. This approach uses our technique mentioned in Section 2.3 to classify queries into r categories. It then assigns the j-th replica of each data segment to a worker in a manner optimized for the j-th query category. For example, assume two query categories which represent queries on a single graph snapshot (Category I) and queries on all snapshots (Category II), respectively. Then, each graph snapshot replica for Category I should be distributed over all workers to parallelize queries to the maximum extent. On the other hand, each graph snapshot replica for Category II should 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).

2.6 Query-Aware Replica Selection

When a query is submitted to the G* system, the master determines into which query category, among the r categories, the submitted query best fits according to the metric described in Section 2.3. Then it runs the query on the graph snapshot replicas that are optimized for the chosen query category. For example, queries on a single graph snapshot (Category I) should be executed on the graph snapshot replica for Category I (i.e., the replica distributed over all workers).

3. DEMONSTRATION DETAILS

This section describes the demonstration environment (Section 3.1), demonstration interface (Section 3.2), and demonstration scenarios (Section 3.3).

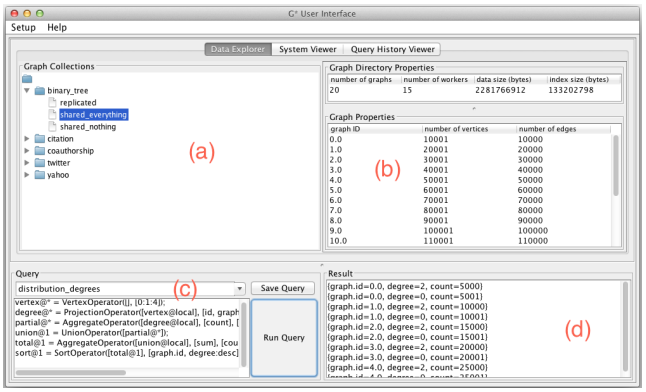


Figure 3: The G* User Interface

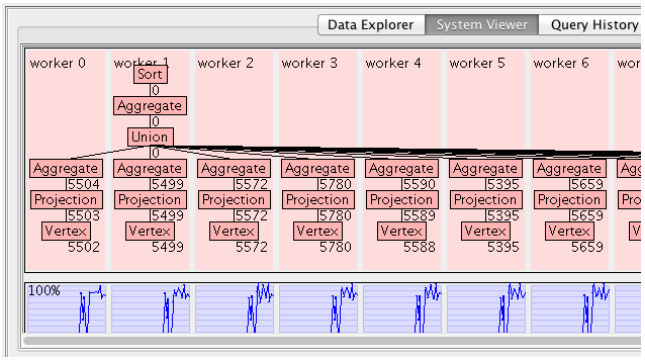


Figure 4: G* System Viewer

3.1 Demonstration Environment

We will prepare a cluster of four Mac Minis, each of which has a 2.3GHz Quad-Core Intel i7 CPU, 8GB RAM, and a 1TB Serial ATA Drive. To fully utilize this cluster's 16 CPU cores, we will run one G* master and 15 worker servers.

Data. We will provide series of graph snapshots that represent large, evolving networks at different points in time. We will construct these graph snapshots using Twitter messages [13], network traffic records between Yahoo! servers and the rest of the world [14], citation and coauthorship networks [12], and a binary tree generator. We will prepare multiple distribution/replication configurations for each series of graph snapshots (Sections 3.3.1 and 3.3.4) and show their impact on query completion time.

Queries. We will demonstrate the benefits of our techniques mentioned in Section 2 by using queries on a set of selected graph snapshots. These queries will find, from each graph snapshot, certain aspects including:

- the average degree
- the clustering coefficient distribution
- the PageRank distribution
- the centrality of a vertex
- the size of the largest component
- the k vertices with the largest increase in degree.

Further details of these queries are available at:
`http://www.cs.albany.edu/~gstar/quick-start-guide`.

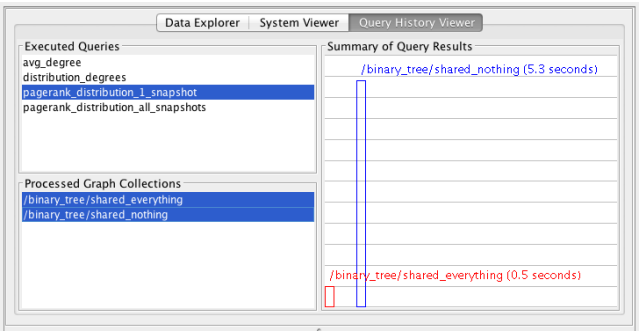


Figure 5: G* Query History Viewer

3.2 Demonstration Interface

Our demonstration will proceed using the graphical user interface shown in Figures 3, 4, and 5.

The *Data Explorer* (Figure 3) shows all of the prepared graph snapshots in a hierarchical fashion (a). When a series of graph snapshots is selected, the details of these snapshots are displayed in the *Graph Properties* panel (b).

The *Query* panel (Figure 3 (c)) allows users to enter queries and then run them on the series of graph snapshots chosen in the *Data Explorer*. The *Result* panel (d) shows query results.

The *System Viewer* (Figure 4) displays the execution of a query on G* workers. This viewer shows the number of data objects that each operator has processed as well as the the resource usage of each G* worker.

The *Query History Viewer* (Figure 5) keeps track of the queries that have been executed and allows users to selectively compare the running time of these queries. Our demonstrations (Section 3.3) will use this viewer to show the effectiveness of our techniques described in Section 2.

3.3 Demonstration Scenarios

We will provide the following demonstrations:

3.3.1 Impact of Snapshot Distribution

We will prepare each series of graph snapshots in both *shared-nothing* (each graph is stored on one distinct worker) and *shared-everything* (each graph is evenly distributed over all workers) configurations. We will then show how the effects of these configurations on query execution time vary depending on the type of query being run. In particular, we will demonstrate that:

- When a single graph snapshot is queried, *shared-everything* is more advantageous than *shared-nothing* due to the simultaneous use of workers.
- When graph snapshots are similarly sized and all of them are queried together, *shared-nothing* is preferable to *shared-everything* since the former incurs substantially lower network overhead while both configurations enable parallel query execution on all workers.

3.3.2 Benefits of Segment Swapping

Our segment swapping technique in Section 2.2 gradually exchanges segments between G* workers to reduce the expected query time. We will verify the effectiveness of this technique by preparing an inappropriate distribution configuration (e.g., *shared-nothing* when queries on a single graph snapshot are popular) and then show how G* workers exchange segments to accelerate popular queries.

3.3.3 Effectiveness of Query Pattern Identification

Our query pattern identification technique (Section 2.3) groups queries into k patterns based on the graph segments that these queries access. We will show the graph segments that have been queried during the demonstration session as well as the k query patterns obtained from our technique.

3.3.4 Advantages of Snapshot Replication

We will demonstrate how G* distributes the replicas of each graph snapshot over G* workers (Section 2.5). We will also show that G* can select, for each query, the most advantageous replica configuration (e.g., *shared-everything* for a PageRank query on a single graph snapshot and *shared-nothing* for a PageRank query on all graph snapshots).

4. REFERENCES

[1] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. PageRank on an Evolving Graph. In *KDD*, pages 24–32, 2012.

[2] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[4] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. Gamma - A High Performance Dataflow Database Machine. In *VLDB*, pages 228–237, 1986.

[5] J.-H. Hwang, J. Birnbaum, A. Labouseur, P. W. Olsen Jr., S. R. Spillane, J. Vijayan, and W.-S. Han. G*: A System for Efficiently Managing Large Graphs. Technical Report SUNYA-CS-12-04, CS Department, University at Albany – SUNY, 2012.

[6] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. In *SC*, page 29, 1995.

[7] A. Labouseur, P. W. Olsen Jr., and J.-H. Hwang. Scalable and Robust Management of Dynamic Graph Data. In *BD*^{3}*@VLDB*, pages 43–48, 2013.

[8] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[9] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. Technical Report TR 00-018, Computer Science and Engineering, U. of Minnesota, 2000.

[10] Z. Shang and J. X. Yu. Catch the Wind: Graph Workload Balancing on Cloud. In *ICDE*, pages 553–564, 2013.

[11] S. R. Spillane, J. Birnbaum, D. Bokser, D. Kemp, A. Labouseur, P. W. Olsen Jr., J. Vijayan, and J.-H. Hwang. A Demonstration of the G* Graph Database System. In *ICDE*, pages 1356–1359, 2013.

[12] Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data/`.

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

[14] Yahoo! Network Flows Data. `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`.