Scalable and Robust Management of Dynamic Graph Data[*]

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

ABSTRACT

Most real-world networks evolve over time. This evolution can be modeled as a series of graphs that represent a network at different points in time. Our G* system enables efficient storage and querying of these *graph snapshots* by taking advantage of the commonalities among them. We are extending G* for highly scalable and robust operation.

This paper shows that the classic challenges of data distribution and replication are imbued with renewed significance given continuously generated graph snapshots. Our data distribution technique adjusts the set of worker servers for storing each graph snapshot in a manner optimized for popular queries. Our data replication approach maintains each snapshot replica on a different number of workers, making available the most efficient replica configurations for different types of queries.

1. INTRODUCTION

Real-world networks, including social networks and the Web, constantly evolve over time [3]. Periodic snapshots of such a network can be represented as graphs where vertices represent entities and edges represent relationships between entities. These *graph snapshots* allow us to analyze the evolution of a network over time by examining variations of certain features, such as the distribution of vertex degrees and clustering coefficients [17], network density [20], the size of each connected component [17, 18], the shortest distance between pairs of vertices [20, 23], and the centrality or eccentricity of vertices [23]. Trends discovered by these analyses play a crucial role in sociopolitical science, marketing, security, transportation, epidemiology, and many other areas. For example, when vertices represent people, credit cards, and consumer goods, and edges represent ownership and purchasing relationships, disruptions in degree distribution, viewed over time, may indicate anomalous behavior, perhaps even fraud.

Several single-graph systems are available today: Google's Pregel [21], Microsoft's Trinity [29], Stanford's GPS [24], the open source Neo4j [22], and others [2, 5, 6, 7, 12, 14]. They, however, lack support for efficiently managing large graph snapshots. Our G* system [13, 27] efficiently stores and queries graph snapshots on multiple worker servers by taking advantage of the commonalities among snapshots. DeltaGraph [16] achieves a similar goal. Our work is complementary to DeltaGraph in that it focuses on new challenges in data distribution and robustness in the context of continuously creating large graph snapshots.

Single-graph systems typically distribute the entirety of a single graph over all workers to maximize the benefits of parallelism. When there are multiple graph snapshots, however, distributing each snapshot on all workers may slow down query execution. In particular, if multiple snapshots are usually queried together, it is more advantageous to *store each snapshot on fewer workers* as long as the overall queried data are balanced over all workers. In this way, the system can *reduce network overhead* (i.e., improve query speed) while *benefiting from high degrees of parallelism.* We present a technique that automatically adjusts the number of workers in a manner optimized for popular queries.

As implied above, there are vast differences in execution time depending on the distribution configurations and the number of snapshots queried together. Replication gives us, in addition to enhanced system reliability, the opportunity to utilize as many distribution configurations as there are replicas. G* constructs r replicas for each snapshot to tolerate up to $r-1$ simultaneous worker failures. Our technique classifies queries into r categories and optimizes the distribution of each replica for one of the query categories.

In this paper, we make the following contributions:

- We define the problem of distributing graph snapshots and present a solution that expedites queries by adjusting the set of workers for storing each snapshot.
- We provide a technique for adaptively determining replica placement to improve system performance and reliability.
- We present preliminary evaluation results that show the effectiveness of the above techniques.
- We discuss our research plans to complete the construction of a highly scalable and reliable system for managing large graph snapshots.

The remainder of the paper is organized as follows: Section 2 presents the research context and provides formal definitions of the problems studied in the paper. Sections 3 and 4 describe our new techniques for distributing and replicating graph snapshots. Section 5 presents our preliminary evaluation results. Section 6 discusses related work. Section 7 concludes this paper.

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

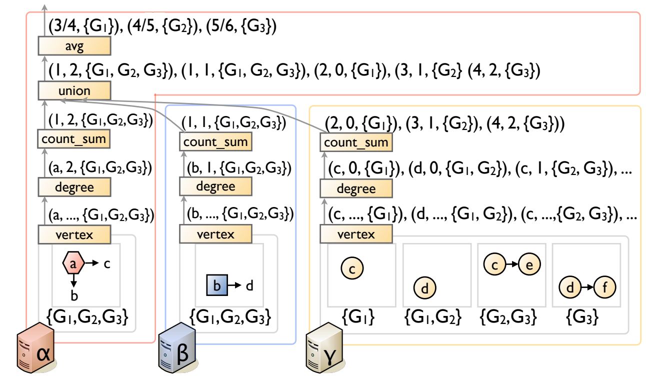

(3/4, {G1}), (4/5, {G2}), (5/6, {G3})
avg
(1, 2, {G1, G2, G3}), (1, 1, {G1, G2, G3}), (2, 0, {G1}), (3, 1, {G2} (4, 2, {G3})
union
(1, 2, {G1,G2,G3})
count_sum
(a, 2, {G1,G2,G3})
degree
(a, ..., {G1,G2,G3})
vertex
{G1,G2,G3}
(1, 1, {G1,G2,G3})
count_sum
(b, 1, {G1,G2,G3})
degree
(b, ..., {G1,G2,G3})
vertex
{G1,G2,G3}
(2, 0, {G1}), (3, 1, {G2}), (4, 2, {G3}))
count_sum
(c, 0, {G1}), (d, 0, {G1, G2}), (c, 1, {G2, G3}), ...
degree
(c, ..., {G1}), (d, ..., {G1, G2}), (c, ...,{G2, G3}), ...
vertex
{G1} {G1,G2} {G2,G3} {G3}
α β γ

Figure 1: Parallel Calculation of Average Degree

2. BACKGROUND

2.1 Summary of G*

G* is a distributed system for managing large graph snapshots that represent an evolving network at different points in time [13, 27]. As Figure 1 shows, these graph snapshots (e.g., G_1, G_2, and G_3) are distributed over *workers* (e.g., α, β, and γ) that both store and query the graph data assigned to them. The *master* of the system (not shown in Figure 1) transforms each submitted query into a network of operators that process graph data on workers in a parallel fashion. Our previous work on G* can be summarized as follows:

Graph Storage. In G*, each worker efficiently stores its data by taking advantage of commonalities among graph snapshots. Figure 2 shows how worker γ from Figure 1 incrementally stores its portion of snapshots G_1, G_2, and G_3 on disk. The worker stores c_1 and d_1, the first versions of c and d, when it stores G_1. When vertex c obtains a new edge to e in G_2, the worker stores c_2, the second version of c, which shares commonalities with the previous version and also contains a new edge to e. When vertex d obtains a new edge to f in G_3, the worker stores d_2, the second version of d which contains a new edge to f. All of these vertex versions are stored on disk only once regardless of how many graph snapshots they belong to.

To track all of these vertex versions, each worker maintains a *Compact Graph Index* (*CGI*) that maps each combination of vertex ID and graph ID onto the disk location that stores the corresponding vertex version. For each vertex version (e.g., c_2), the CGI stores only one (*vertex ID, disk location*) pair in a collection for the combination of snapshots that contain that vertex version (e.g., $\{G_2, G_3\}$). In this manner, the CGI handles only vertex IDs and disk locations while all of the vertex and edge attributes are stored on disk. Therefore, the CGI can be kept fully or mostly in memory, enabling fast lookups and updates. To prevent the CGI from becoming overburdened by managing too many snapshot combinations, each worker automatically groups snapshots and then separately indexes each group of snapshots [13, 27].

Query Processing. Like traditional database systems, G* supports sophisticated queries using a dataflow approach where operators process data in parallel. To quickly process queries on multiple graph snapshots, however, G* supports special operators that share computations across snapshots.

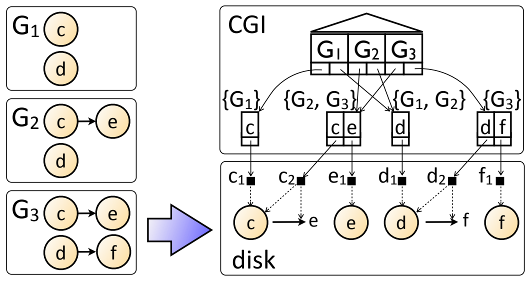


Figure 2: Storage of Snapshots G_1, G_2, G_3 and CGI

PageRank Query	Shared-Nothing	Shared-Everything
One snapshot	285 seconds	22 seconds
All snapshots	285 seconds	2,205 seconds

Table 1: Impact of Graph Snapshot Configuration

Figure 1 shows how the average degree calculation takes place in parallel over three workers. The `vertex` and `degree` operators in Figure 1 compute the degree of each vertex only once while associating the result with all of the relevant graph snapshots (e.g., the degree of vertex a is shared across G_1, G_2, and G_3). In the example, the `count_sum` operators aggregate the degree data, the `union` operator merges these data, and the `avg` operator produces the final result. Details of our graph processing operators and programming primitives for easy implementation of custom operators are provided in our earlier papers [13, 27].

2.2 Problem Statements

Our previous work [13, 27] focused on efficiently storing and querying graph snapshots. We now take up the challenge of doing so in a highly scalable and robust manner.

2.2.1 *Multiple Snapshot Distribution*

Accelerating computation by distributing data over multiple servers has been a popular approach in parallel databases [10] and distributed systems [9]. Furthermore, techniques for partitioning graphs to facilitate parallel computation have also been developed [15, 24, 25, 26]. 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.

Example. Consider a scenario where each of 100 similarly-sized graph snapshots contains approximately 1 million vertices and 100 million edges. Assume also that the system consists of one master and 100 workers. Table 1 compares two snapshot distribution configurations: *Shared-Nothing*, where each of the 100 snapshots is stored on one distinct worker, and *Shared-Everything*, where each snapshot is evenly distributed over all of the 100 workers. For each of these configurations, two types of queries for computing the PageRank of each vertex are executed: *Query One Snap-*

shot, and *Query All Snapshots*. The explanations below are based on our evaluation results (see Section 5 for details).

In the case of *Shared-Nothing*, querying one snapshot using only one worker takes 285 seconds (205 seconds to construct the snapshot from disk and 80 seconds to run 20 iterations of PageRank). Querying all snapshots on all workers in parallel 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 80/100 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*).

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 (Section 2.1) and *popular snapshots* for query speed. In this paper, 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 some or all of the snapshots, find a distribution* $\{V_{i,w} : i = 1, 2, \cdots \ \wedge \ w = 1, 2, \cdots, n\}$ *that minimizes* $\sum_{q \in Q} time(q, \{V_{i,w}\})$ *where* $V_{i,w}$ *denotes the set of vertices that are from snapshot* $G_i(V_i, E_i)$ *and that are assigned to worker w, and* $time(q, \{V_{i,w}\})$ *represents the execution time of query* $q \in Q$ *on the distributed snapshots* $\{V_{i,w}\}$ *satisfying (1)* $\cup_{w=1}^{n} V_{i,w} = V_i$ *(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).*

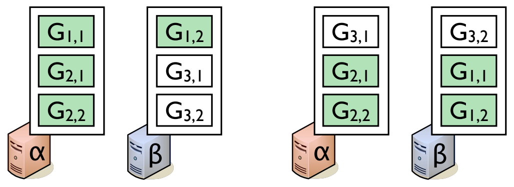
Our solution to the above problem is presented in Section 3.

2.2.2 *Snapshot Replication*

There have been various techniques for replicating data to improve availability and access speed [8, 11, 28]. 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:

PROBLEM 2. **(Snapshot Replication)** *Given a series of graph snapshots* $\{G_i(V_i, E_i) : i = 1, 2, \cdots\}$*, the degree of replication r, n workers, and a set of queries Q on some or all of the snapshots, find a replica distribution* $\{V_{i,j,w} : i = 1, 2, \cdots \ \wedge \ j = 1, 2, \cdots, r \ \wedge \ w = 1, 2, \cdots, n\}$ *that minimizes* $\sum_{q \in Q} time(q, \{V_{i,j,w}\})$ *where* $V_{i,j,w}$ *denotes the set of vertices that are from the jth 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* $time(q, \{V_{i,j,w}\})$ *denotes the execution time of query q on the distributed snapshot replicas* $\{V_{i,j,w}\}$ *satisfying (1)* $\cup_{w=1}^{n} V_{i,j,w} = V_{i,j} = V_i$ *for* $j = 1, 2, \cdots, r$*, (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 edges, which tolerates* $r - 1$ *simultaneous worker failures).*

Section 4 presents our solution to the above problem.



(a) Before Exchange (b) After Exchange

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

3. GRAPH SNAPSHOT DISTRIBUTION

As mentioned in Section 2.2.1, G* needs to store each graph snapshot on an appropriate number of workers while balancing the utilization of network and CPU resources. In contrast to traditional methods for partitioning a *static* graph [15, 25], G* must determine the location of each vertex and its edges on the fly in response to a continuous influx of data from external sources.

Our *dynamic* data distribution approach meets the above requirements. In this approach, each G* worker partitions its graph data into *segments* with a certain maximum size (e.g., 10GB) so that it can control its load by migrating some segments to other workers (Section 3.1). Our approach continuously routes incoming messages for updating vertices and edges to appropriate workers with low latency (Section 3.2). 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 3.3). It does all of the above while supporting G*'s graph processing operators (Section 3.4).

3.1 Load Balancing

In G*, each worker periodically communicates with a randomly chosen worker to balance graph data. Our key principles in load balancing are to (1) *maximize the benefits of parallelism by uniformly distributing data that are queried together* and (2) *minimize network overhead by co-locating data from the same snapshot.* Consider Figure 3(a) where three snapshots are partitioned into a total of 6 similarly-sized segments. In this example, each of workers α and β are assigned a segment from snapshot G_1, α is assigned two segments from G_2, and β is assigned two segments from G_3. If snapshots G_1 and G_2 are frequently queried together (see those shaded in Figure 3(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 $G_{1,2}$. This problem can be remedied by exchanging $G_{1,1}$ and $G_{3,1}$ between the workers, which results in a balanced distribution of the data queried together (i.e., G_1 and G_2) and localized processing of G_1 on β and G_2 on α, respectively.

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 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 $\sum_{q \in \mathcal{Q}_k} p(q) \cdot time(q, S_i, S_j)$ where $\mathcal{Q}_k$ 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 placements S_i and S_j.

Our technique obtains $\mathcal{Q}_k$ (equivalently, k popular combinations of segments queried together) as follows: Sort segments from $S_i \cup S_j$ in order of decreasing popularity. Initialize $\mathcal{Q}_k$ (for storing k popular query patterns) with the first segment. Then, for each of the remaining segments, combine it with each element from $\mathcal{Q}_k$ and insert the result back into $\mathcal{Q}_k$. Whenever $|\mathcal{Q}_k| > k$, remove its least popular element. We estimate the popularity of each combination of segments by consolidating the counting synopses [4] for those segments. Whenever a query accesses a segment, the associated synopsis is updated using the ID of the query.

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.

3.2 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 (*vertex ID, segment ID*) pair in an index similar to the CGI (Section 2.1). 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 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.

3.3 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 [15]. 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 2.1). 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 3.2.

3.4 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 [13]. 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 3.2).

4. GRAPH SNAPSHOT REPLICATION

G* masks up to $r-1$ simultaneous worker failures by creating r copies of each graph data segment. As discussed in Sections 2.2 and 3, 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 [1] to classify queries into r categories based on the number of graphs that they access. It then assigns the j-th replica of each data segment to a worker in a manner optimized for the j-th query category. The master and workers support this approach as follows:

4.1 Updates of Vertices and Edges

Updates of vertices and edges are handled as described in Section 3.2 except that they are 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 $(w+1)\%n, (w+2)\%n, \cdots, (w+r-1)\%n$ where n denotes the number of workers. If a worker fails, the master assigns another worker to take over.

4.2 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.

4.3 Query-Aware Replica Selection

For each query, the master identifies the worker locations of the data segment replicas to process. To this end, the

Message passing (12-bytes/message)	1M messages/sec
Disk I/O bandwidth	200 Mbytes/sec
Snapshot construction in memory	200 seconds
PageRank iteration per snapshot	4 seconds

Table 2: Speed and Bandwidth Observations

# Cores	1	2	4	8	16	24	48
Speedup	1.0	1.9	3.7	5.9	9.7	12.5	14.7

Table 3: Actual Speedup Result

master 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. Next, as Figure 1 shows, the master instantiates operators on these workers and starts executing the query.

4.4 Load Balancing

Each worker balances its graph data as explained in Section 3. The only difference 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 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.

5. PRELIMINARY EVALUATION

This section presents our preliminary results obtained by running G* on a six-node, 48-core cluster. In this cluster, each machine has two Quad-Core Xeon E5430 2.67 GHz CPUs, 16GB RAM, and a 2TB hard drive. We plan to extend these experiments with more queries on larger data sets in a bigger cluster (Section 7).

To construct a realistic example in Section 2.2.1, we measured the overhead of key operations summarized in Table 2. In our evaluation, a worker was able to transmit up to 1 million messages to other workers within a second, although a 1Gbps 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 in Java. Furthermore, reading approximately 1Gbytes 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.

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 3 shows, for the shortest distance query, the speedup achieved by distributing the snapshots over more workers. The highest speedup was achieved with 48 workers. This table also

SSSP Query	All Workers	Subset of Workers
One snapshot	8.2 seconds	19.2 seconds
All snapshots	80.5 seconds	53.2 seconds

Table 4: Impact of Graph Data Distribution

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 creating large graph snapshots.

The effectiveness of two different distributions is demonstrated in Table 4. 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 4 than Table 1 due to a smaller number of message transmissions and fewer workers. Table 4 also demonstrates the benefit of G* in executing queries 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 across smaller snapshots.

6. RELATED WORK

In this section, we briefly summarize related research, focusing on previous graph systems, data distribution, and data replication.

Previous Graph Systems. In contrast to systems which process one graph at a time [2, 5, 6, 7, 12, 14, 21, 22, 24, 29], G* efficiently executes sophisticated queries on multiple graph snapshots. G*'s benefits over previous systems are experimentally demonstrated in our prior work [13]. DeltaGraph [16] and GraphChi [19] are promising systems for dynamic graphs but do not directly address the data distribution/replication issues considered in this paper.

Data Distribution. Traditional graph partitioning techniques split a static graph into subgraphs in a manner that minimizes the number of crossing edges [15, 25]. There are also recent graph repartitioning schemes that observe communication patterns and then move vertices to reduce network overhead [24, 26]. In contrast to them, 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 (Section 3).

Data Replication. There has been extensive work on data replication that focused on improving data availability and performance [8, 11, 28]. Researchers developed techniques for ensuring replica consistency [11] and finding most advantageous replica placement [8]. Stonebraker et al. proposed an approach that stores each database replica differently, optimized for a different query type [28]. 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 types of queries.

7. CONCLUSIONS AND FUTURE WORK

We presented G*, a scalable and robust system for storing and querying large graph snapshots. G* tackles new data distribution and replication challenges that arise in the context of continuously creating large graph snapshots. Our data distribution technique efficiently stores graph data on the fly using multiple worker servers in parallel. 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 maintains each graph replica on a different number of workers, making available the most efficient storage configurations for various combinations of queries.

We are working on full implementations of the techniques presented in this paper to enable new experiments with additional queries on larger data sets. We will analyze these techniques to classify their complexity. We plan to look into the challenges of scheduling groups of queries, dealing with varying degrees of parallelism, resource utilization, and user-generated performance preferences. We are exploring failure recovery techniques for long-running queries while exposing the tradeoff between recovery speed and execution time. We also want to study opportunities for more granular splitting, merging, and exchanging of data at the vertex and edge level rather than in large segments as discussed in this paper. We intend to seek opportunities for gains in execution speed at the expense of storage space by segregating recent and popular "hot" data (which we could store in a less compressed manner) from less popular "cold" data (which could be highly compressed).

8. REFERENCES

[1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A Framework for Clustering Evolving Data Streams. In *VLDB*, pages 81–92, 2003.

[2] Apache Hama. *http://hama.apache.org*.

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

[4] K. S. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On Synopses for Distinct-Value Estimation Under Multiset Operations. In *SIGMOD*, pages 199–210, 2007.

[5] Cassovary. Open Sourced from Twitter *https://github.com/twitter/cassovary*.

[6] A. Chan, F. K. H. A. Dehne, and R. Taylor. CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *IJHPCA*, 19(1):81–97, 2005.

[7] R. Chen, X. Weng, B. He, and M. Yang. Large Graph Processing in the Cloud. In *SIGMOD*, pages 1123–1126, 2010.

[8] Y. Chen, R. H. Katz, and J. Kubiatowicz. Dynamic Replica Placement for Scalable Content Delivery. In *IPTPS*, pages 306–318, 2002.

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

[10] 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.

[11] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, pages 173–182, 1996.

[12] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *POOSC*, 2005.

[13] 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.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System. In *ICDM*, pages 229–238, 2009.

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

[16] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. *CoRR*, abs/1207.5777, 2012.

[17] G. Kossinets and D. J. Watts. Empirical Analysis of an Evolving Social Network. *Science*, 311(5757):88–90, 2006.

[18] R. Kumar, J. Novak, and A. Tomkins. Structure and Evolution of Online Social Networks. In *KDD*, pages 611–617, 2006.

[19] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *OSDI*, pages 31–46, 2012.

[20] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking diameters and Possible Explanations. In *KDD*, pages 177–187, 2005.

[21] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.

[22] Neo4j The Graph Database. *http://neo4j.org/*.

[23] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On Querying Historical Evolving Graph Sequences. *PVLDB*, 4(11):726–737, 2011.

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

[25] 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.

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

[27] 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.

[28] M. Stonebraker et al. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[29] Trinity. *http://research.microsoft.com/en-us/projects/trinity/*.