A Demonstration of the G* Graph Database System

Sean R. Spillane [#], Jeremy Birnbaum [#], Daniel Bokser [#], Daniel Kemp [#], Alan Labouseur [#],
Paul W. Olsen Jr. [#], Jayadevan Vijayan [#], Jeong-Hyon Hwang [#], Jun-Weon Yoon [*]

[#] *Department of Computer Science, University at Albany – State University of New York*
1400 Washington Avenue, Albany, NY 12222, USA
{seans, jbirn, dbokser, dkemp, alan, polsen, appu, jhh}@cs.albany.edu

[*] *Department of Supercomputing Support, KISTI Supercomputing Center*
335 Gwahangno, Yuseong-gu, Daejeon, 305-806, Republic of Korea
jwyoon@kisti.re.kr

***Abstract*—The world is full of evolving networks, many of which can be represented by a series of large graphs. Neither the current graph processing systems nor database systems can efficiently store and query these graphs due to their lack of support for managing multiple graphs and lack of essential graph querying capabilities. We propose to demonstrate our system, G*, that meets the new challenges of managing multiple graphs and supporting fundamental graph querying capabilities. G* can store graphs on a large number of servers while compressing these graphs based on their commonalities. G* also allows users to easily express queries on graphs and efficiently executes these queries by sharing computations across graphs. During our demonstrations, conference attendees will run various analytic queries on large, practical data sets. These demonstrations will highlight the convenience and performance benefits of G* over existing database and graph processing systems, the effectiveness of sharing in graph data storage and processing, as well as G*'s scalability.**

I. Introduction

Graphs model a vast array of networks: social networks, communications routing systems, road and highway systems, as well as citation and coauthorship networks. These networks evolve naturally over time as friends change, routers crash, bypasses are built, and so on. Many researchers seek to gain insight into this evolution by analyzing a series of graphs that are periodic snapshots of a dynamic network [1], [2], [3].

It is important to note that in many cases each snapshot of a network is considerably similar to its predecessor because evolution tends to occur gradually. For example, the state of LinkedIn today and tomorrow will be substantially similar. Exploiting these commonalities in data storage (see Section II-A) and computation (see Section II-B) enables us to achieve high performance. Another crucial observation is that analysis of large networks typically requires extracting aggregate information from relevant graphs [1], [2], [3] (see also Figures 3 and 5). This analysis in general can be expressed as a sophisticated query that combines graph algorithms and aggregate data operations.

Existing graph processing systems such as Pregel [4], Giraph [5], Trinity [6] and Neo4j [7], and traditional database systems such as PostgreSQL [8], do not meet the above requirements. Current graph processing systems process only a single graph at a time and are not suited to querying sets of graphs. On the other hand, relational database systems must break graph structures into edges that are stored in a relation and thus require expensive join operations for most graph analysis tasks [9]. Furthermore, existing graph processing systems and relational database systems cannot readily take advantage of commonalities among graphs in their storage and processing of data.

To remedy the limitations of these systems, we have developed a new parallel graph database system, G*, that efficiently stores and queries collections of large graphs. G* can store graphs on a large number of servers while compressing these graphs based on the commonalities among them. Furthermore, G* executes sophisticated graph queries using a network of operators that process graph data in parallel. To accelerate queries on multiple graphs, these operators process vertices and edges once and share the results across relevant graphs.

G*'s shared storage and shared computation abilities make it unique among graph processing systems. Our demonstrations will exhibit G*'s unique prowess by showing its:

- performance advantages over relational database systems,
- superiority over current graph processing systems,
- benefits from shared storage and computation over isolated storage and computation, and
- ability to scale.

Our demonstrations will present conference attendees with data from the Twitter social network [10], Yahoo! server logs [11], citation and coauthorship networks [12], and generated binary trees. Attendees will be encouraged to:

- explore and query these data sets by interacting with the Data Explorer tool (Figure 4),
- monitor the execution of their queries using the System Viewer (Figure 6), and
- compare and contrast their results with others using the Query History Viewer (Figure 7).

In the remainder of this demonstration proposal, we present an architectural overview of G* and outline our graph data storage and query processing techniques (Section II). In Section III, we describe the demonstration environment, demonstration interface, and several specific demonstration scenarios.

II. Summary of G*

As Figure 1 illustrates, G* is composed of a collection of servers managed by a *master* server. When a query is

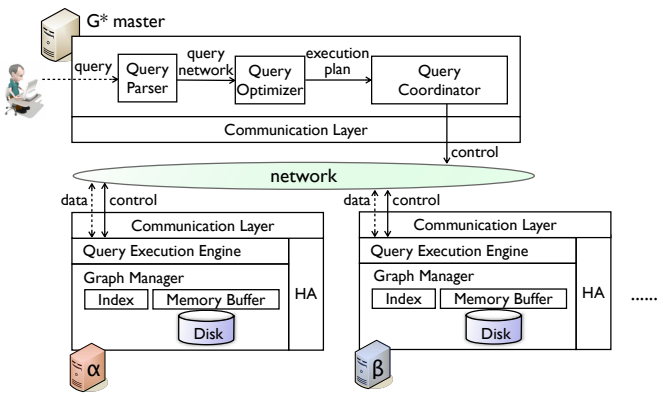

Fig. 1. G* Architecture

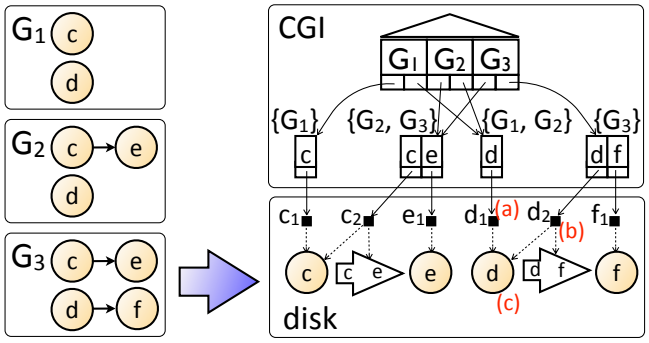

Fig. 2. Compact Storage and Indexing of Graphs G_1, G_2, and G_3

submitted to the master, it is first translated by the *query parser* into an operator network, which in turn is transformed into an optimized execution plan by the *query optimizer*. The *query coordinator* creates and runs operators on other servers via their *query execution engines*. An operator on a server may get data from the server's graph manager, which fetches them from that server's memory and disk, or from other operators (Figure 3). Data are sent between servers via the communication layer. The *high availability* module performs tasks for detecting and recovering from server failures. Section II-A describes how the graph manager on each server efficiently stores and retrieves its portion of the graph data. Section II-B explains how the query coordinator and query execution engines collaborate to efficiently process queries on multiple graphs.

A. Compact Storage of Graphs

G* assigns a vertex and its outgoing edges to the same server for high data locality, which allows the server to access all of a vertex's edges without contacting other servers. When multiple graphs represent a network at different points in time, these graphs may contain a large number of common vertices and edges. Therefore, each G* server tracks the variation of each vertex and its edges while saving one version of them on disk for all of the graphs in which they do not vary. For example, in Figure 2, vertex d remains the same in graphs G_1 and G_2. Therefore, that version of d (denoted as d_1 in Figure 2) is stored only once on disk (see (a)). When vertex

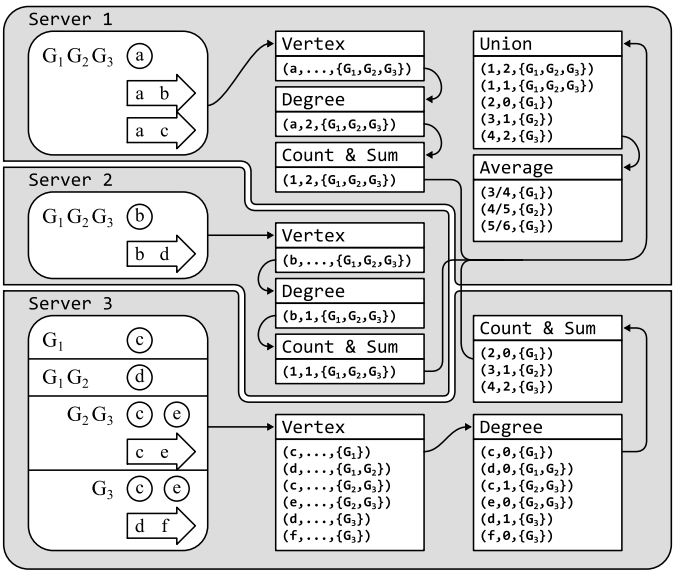

Fig. 3. Average Degree Computation on Graphs G_1, G_2, and G_3

d in graph G_3 obtains an edge to vertex f, a new version of d (denoted as d_2) is stored on disk (b). For space efficiency, vertex versions share common attributes and edges (i.e., only the difference between versions is stored on disk each time). Therefore, in Figure 2, the attribute values of vertex d (e.g., the address of a server or a person) are stored only once (c) and shared across versions d_1 and d_2.

To quickly access the vertex versions which are stored on disk, we developed an index that stores only one *(vertex ID, disk address)* pair for each vertex version. This index, called the Compact Graph Index (CGI), keeps the *(vertex ID, disk address)* pair of a vertex version (e.g., c_2 in Figure 2) in a collection for the combination of graphs that contain that vertex version (e.g., $\{G_2, G_3\}$). Due to the deduplicated storage of vertex IDs, graph IDs, and disk addresses, the CGI can be kept fully or mostly in memory, enabling fast lookups and updates. To limit the overhead of tracking too many graph combinations, we also developed a technique that groups graphs and then separately indexes each group of graphs [13].

B. Efficient Query Execution on a Set of Graphs

Just as we can remove redundancy via shared storage of graph data, we can also remove redundancy via shared processing of graph data. Given a query on a set of graphs, G* constructs an operator network that consumes the vertices and edges from these graphs, filtering and synthesizing data into information (Figure 3). Using the CGI to determine which vertices and edges are shared among these graphs, G* operators process those vertices and edges only once, which eliminates any redundancy in computation.

Figure 3 illustrates the operations involved in taking the average vertex degree for graphs G_1, G_2 and G_3. The `Vertex` operator emits tuples that contain the attribute values of a vertex and the IDs of the graphs which contain that vertex. These tuples are consumed by `Degree`, which emits tuples that contain a vertex's degree and the IDs of the graphs that contain

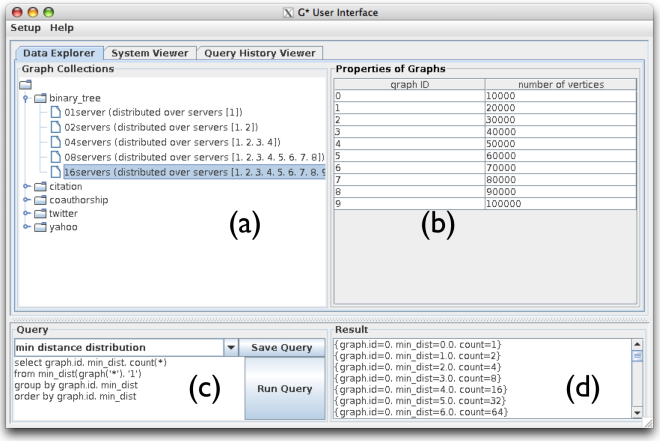

Fig. 4. The G* User Interface

that vertex. Note that each vertex ID in the output may appear more than once, since it will be emitted for each version of that vertex. The output tuples of `Degree` are then consumed by `Count & Sum`, which emits tuples that contain the total number of vertices in a graph, the total number of edges in that graph, and that graph's ID. At this point, the `Union` operator receives tuples generated by `Count & Sum` on every server, and emits those tuples as a single stream. These tuples are then consumed by the `Average` operator, which performs division and emits the resulting average vertex degree for each graph as its output. At no point were computations for any vertex or edge repeated. Details of our graph processing operators and programming primitives for easy implementation of custom operators are presented in our earlier work [13].

III. Demonstration Details

We will exhibit G*'s unique abilities through interactive demonstrations that analyze large, real-world data sets. Conference attendees will run queries through our user interface which also provides succinct visualizations of the system performance and resulting data.

A. System Setup

We will run G* on our 72-core cluster located at the University at Albany, State University of New York. In this cluster, each of nine servers has two Quad-Core Xeon E5430 2.67 GHz CPUs, 16GB RAM, and 2TB of hard drive space. If we cannot connect to our server cluster, we will use several laptops preconfigured as a backup cluster.

Data. As Figure 4 shows (further details in Section III-B), demonstration attendees will be able to query various collections of large graphs that contain up to billions of vertices and edges. We constructed these graphs using Twitter messages [10], network traffic records between Yahoo! servers and the rest of the world [11], citation and coauthorship networks [12], and a binary tree generator. Each collection of graphs represents the dynamics of a network in a different way (e.g., hourly snapshots vs. daily snapshots, cumulative graphs that share vertices and edges with their previous graphs vs. noncumulative graphs).

```
1  -- Q1. average vertex degree
2  select graph.id, avg(degree)
3  from (select graph.id, degree(vertex) as degree
4        from graph('*').vertex)
5  group by graph.id
6
7  -- Q2. geodesic distance distribution
8  select graph.id, min_dist, count(*)
9  from min_dist(graph('*'), '1')
10 group by graph.id, min_dist
```

Fig. 5. Example Queries

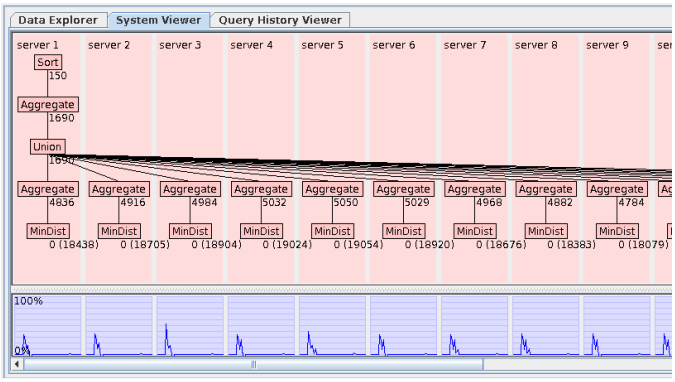

Fig. 6. G* System Viewer

Queries. G* has its own query language, the Declarative Graph Query Language (DGQL), which is modelled after SQL with appropriate extensions for conveniently handling graph data [13]. To help users write queries in this language, we will supply a palette of queries from which they can select and modify a query. Figure 5 shows two example queries which compute, for each graph in a collection, the average vertex degree (Q1) and the distribution of geodesic distances to vertices from vertex '1' (Q2).[1] Other queries include those that find clustering coefficients [1], triadic closure [2], the centrality of vertices [3], and the size of connected components [1].

B. Demonstration Interface

Conference attendees will interact with G* using the user interface shown in Figures 4, 6, and 7.

The *Data Explorer* (Figure 4) provides the ability to select a data set configuration among several (a). Details for these configurations are displayed in the properties panel (b) as they are selected. Attendees will have the opportunity to experiment with many queries that they can select, modify, and execute (c). Real-time results will be shown in their own panel (d). The query (c) and results (d) panels are also present in the System Viewer and Query History Viewer but are left out of Figures 6 and 7 for brevity.

[1] `degree()` on line 3 computes the degree of each vertex. On line 9, `min_dist()` produces a stream of tuples that contain the ID of a vertex and the geodesic distance from vertex '1' to that vertex as the `min_dist` attribute value.

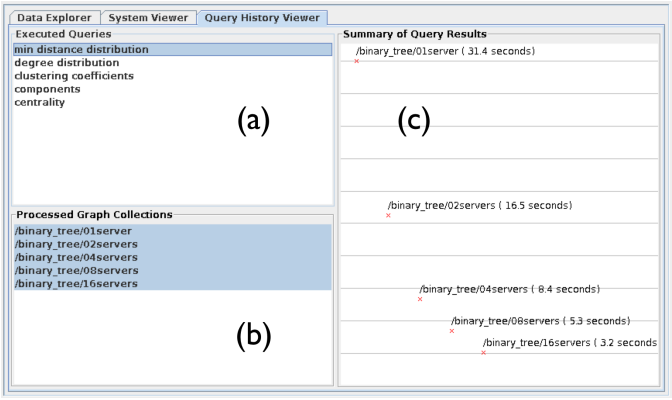

Fig. 7. G* Query History Viewer

The *System Viewer* (Figure 6) lets conference attendees monitor the parallel execution of graph queries. In this example, G* is executing a minimum distance query on multiple servers in parallel. Server 1 is collecting aggregate data from other servers via its `Union` operator. The numeric values displayed under each operator represent the number of tuples that the operator has processed. These values are updated in real-time as execution proceeds. At the same time, attendees can watch each server's resource usage in the graph at the bottom.

The *Query History Viewer* (Figure 7) summarizes the use of G* throughout the demonstration. Real-time updates of queries executed (a) and data sets processed (b) appear in their respective panels while an overall summary (c) is continuously updated.

As illustrated in Figures 4 and 6, our demonstration will allow attendees to have considerable control over G*, its data, and the queries executed upon the data, as well as the ability to watch over the execution in real-time to see G*'s benefits of shared storage and computations. Using the *Query History Viewer* (Figure 7), attendees will also note how G* has performed across different configurations.

C. Demonstration Scenarios

We will provide demonstrations that illustrate G*'s advantages over traditional database systems and graph processing systems. Other demonstrations will highlight the effectiveness of shared storage and computation, as well as G*'s scalability.

Comparison to PostgreSQL. We will demonstrate that relational database systems are neither convenient nor perform well when they process large graphs. We will show how common graph queries like those that compute vertex degrees or clustering coefficients can be simply expressed in G*'s DGQL but not as simply in SQL. In this demonstration, G* will also outperform PostgreSQL in executing these queries since G* can share computations across graphs while PostgreSQL cannot.

Comparison to Pregel-like Systems. In this demonstration, we will compare the performance of G* and Giraph [5], an open-source version of Google's Pregel [4] by solving the single-source shortest path (SSSP) problem on a series of 10 cumulative graphs, each containing 10,000 more edges than its previous one. We will show that G* is substantially faster than Pregel-like systems due to its unique benefit of sharing computations across graphs. We will also show that G* users need to write only a concise DGQL query (Figures 4 and 5) while Pregel-like systems, in stark contrast, require their users to write a lengthy program using a conventional programming language.

Benefits of Shared Storage and Computation. To highlight the benefits of shared storage, we will have our data sets available in both shared and isolated storage configurations, which exploit or ignore commonalities among graphs, respectively. A simple size comparison will demonstrate the efficacy of shared (i.e., de-duplicated) storage over isolated (i.e., redundant) storage. By selecting one or the other of these data sets and executing a query, attendees will experience the difference in processing speed due to the difference in I/O overhead related to data size, thus discovering another benefit of shared storage. Furthermore, attendees will run queries on the shared storage data set in either shared processing or isolated processing modes. In this way, we will demonstrate the benefit of shared computation.

Scalability. Attendees will investigate G*'s ability to scale by running the same query on multiple configurations in which a different number of servers store identical graph data (Figure 4). A few runs of the same query on the same data, but each on a different number of servers, will allow attendees to discover a scaling trend using the Query History Viewer (Figure 7).

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under CAREER Award IIS-1149372 and by the KISTI Supercomputing Center.

REFERENCES

[1] G. Kossinets and D. J. Watts, "Empirical Analysis of an Evolving Social Network," *Science*, vol. 311, no. 5757, pp. 88–90, 2006.
[2] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins, "Microscopic Evolution of Social Networks," in *KDD*, 2008, pp. 462–470.
[3] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On Querying Historical Evolving Graph Sequences," *PVLDB*, vol. 4, no. 11, pp. 726–737, 2011.
[4] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *SIGMOD*. ACM, 2010, pp. 135–146.
[5] Apache Giraph, http://giraph.apache.org/.
[6] Trinity, http://research.microsoft.com/en-us/projects/trinity/.
[7] Neo4j the graph database, http://neo4j.org/.
[8] PostgreSQL, http://www.postgresql.org/.
[9] P. Zhao and J. Han, "On graph query optimization in large networks," *PVLDB*, vol. 3, no. 1, pp. 340–351, 2010.
[10] Twitter Streaming API, https://dev.twitter.com/docs/streaming-api/methods.
[11] Yahoo! Network Flows Data, http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.
[12] Stanford Large Network Dataset Collection, http://snap.stanford.edu/data/.
[13] J.-H. Hwang, J. Birnbaum, A. Labouseur, P. W. O. Jr., S. R. Spillane, J. Vijayan, and W.-S. Han, "G*: A System for Efficiently Managing Large Graphs," CS Department, University at Albany – SUNY, Tech. Rep. SUNYA-CS-12-04, 2012.