

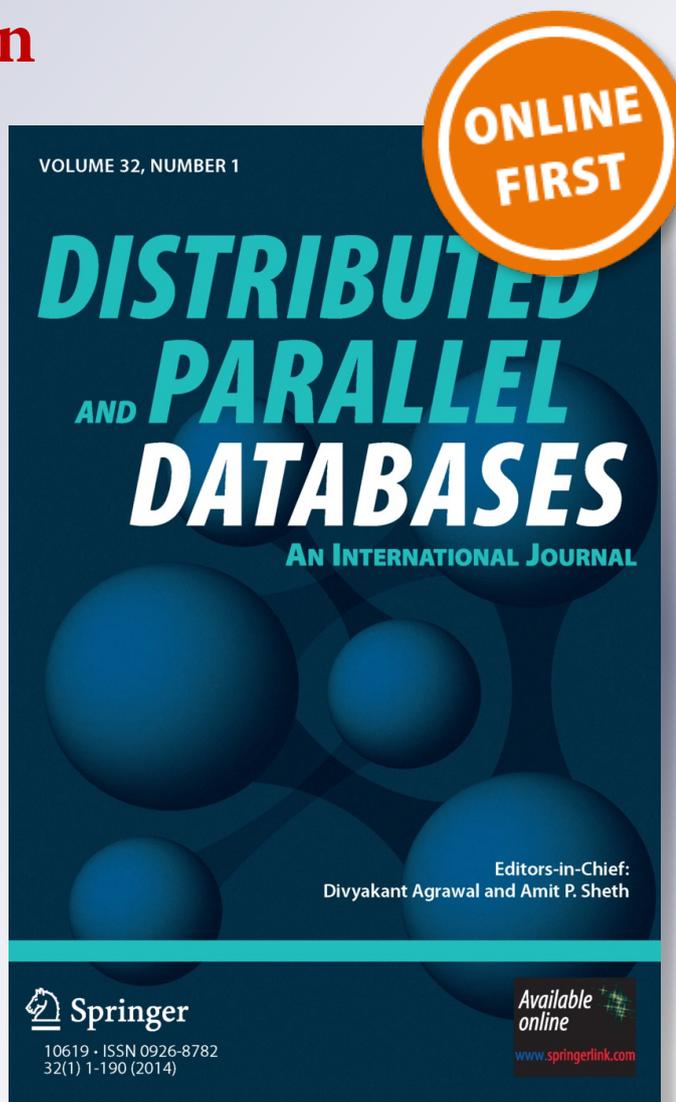
The G^ graph database: efficiently managing large distributed dynamic graphs*

**Alan G. Labouseur, Jeremy Birnbaum,
Paul W. Olsen, Sean R. Spillane,
Jayadevan Vijayan, Jeong-Hyon Hwang
& Wook-Shin Han**

Distributed and Parallel Databases
An International Journal

ISSN 0926-8782

Distrib Parallel Databases
DOI 10.1007/s10619-014-7140-3



Your article is protected by copyright and all rights are held exclusively by Springer Science +Business Media New York. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

The G* graph database: efficiently managing large distributed dynamic graphs

Alan G. Labouseur · Jeremy Birnbaum ·
Paul W. Olsen Jr. · Sean R. Spillane ·
Jayadevan Vijayan · Jeong-Hyon Hwang ·
Wook-Shin Han

© Springer Science+Business Media New York 2014

Abstract From sensor networks to transportation infrastructure to social networks, we are awash in data. Many of these real-world networks tend to be large (“big data”) and dynamic, evolving over time. Their evolution can be modeled as a series of graphs. Traditional systems that store and analyze one graph at a time cannot effectively handle the complexity and subtlety inherent in dynamic graphs. Modern analytics

Communicated by Haixun Wang and Jeffrey Xu Yu.

A. G. Labouseur (✉) · J. Birnbaum · P. W. Olsen Jr. · S. R. Spillane · J. Vijayan · J.-H. Hwang
Department of Computer Science, State University of New York, Albany, NY, USA
e-mail: alan@cs.albany.edu

J. Birnbaum
e-mail: jbirn@cs.albany.edu

P. W. Olsen Jr.
e-mail: polsen@cs.albany.edu

S. R. Spillane
e-mail: seans@cs.albany.edu

J. Vijayan
e-mail: appu@cs.albany.edu

J.-H. Hwang
e-mail: jhh@cs.albany.edu

W.-S. Han
Department of Creative IT Engineering, Pohang University of Science and Technology,
Pohang, Korea
e-mail: wshan@postech.ac.kr

W.-S. Han
Department of Computer Science and Engineering, Pohang University of Science
and Technology, Pohang, Korea

require systems capable of storing and processing series of graphs. We present such a system. G* compresses dynamic graph data based on commonalities among the graphs in the series for deduplicated storage on multiple servers. In addition to the obvious space-saving advantage, large-scale graph processing tends to be I/O bound, so faster reads from and writes to stable storage enable faster results. Unlike traditional database and graph processing systems, G* executes complex queries on large graphs using distributed operators to process graph data in parallel. It speeds up queries on multiple graphs by processing graph commonalities only once and sharing the results across relevant graphs. This architecture not only provides scalability, but since G* is not limited to processing only what is available in RAM, its analysis capabilities are far greater than other systems which are limited to what they can hold in memory. This paper presents G*'s design and implementation principles along with evaluation results that document its unique benefits over traditional graph processing systems.

Keywords Graphs · Queries · Distributed databases · Parallel computing · Big data

1 Introduction

Modern advancements in technology enable access to huge amounts of data from various complex networks including social networks [42,49], transportation networks [42], wireless networks [31], and many others. These networks can be expressed naturally as *graphs* in which *vertices* represent entities and *edges* represent relationships between entities.

Most real-world networks change over time. For example, while today's LinkedIn is substantially similar to yesterday's LinkedIn, they are not identical due to the addition of new users, connections between users, and jobs offered. Social networks, financial networks, citation networks, and road networks all evolve over time. Therefore, there is significant interest in understanding their evolution by extracting certain features from a series of graphs that represent a network at different points in time. These features include the distribution of vertex degrees and clustering coefficients [21], network density [27], triadic closure [26], the size of each connected component [21,23,33], the shortest distance between pairs of vertices [8,27,38], subgraphs representing congested regions [6], the centrality or eccentricity of vertices [38,39], and others [9,25,34,45]. Understanding dynamic graph evolution enables applications in social media analysis [39,44,45], national security, marketing, transportation [6], network management [8,31], epidemiology [34], pharmacology, and more. All of these applications require "connecting the dots" among a great many data points, one of the main challenges in large dynamic graph management.

There are many challenges in managing collections of graphs that represent billions of entities and connections. First, a cluster of servers such as a public/private cloud must be effectively used to store and process the massive amounts of graph data. Second, graphs that represent different times in a dynamically evolving network (e.g., cumulative snapshots of a friendship or sensor network) may be substantially similar to each other. It is therefore crucial to take advantage of those similarities to avoid redundant storage and enable shared processing. Third, finding trends in network evolution requires a combination of graph processing (e.g., finding the

shortest distance between vertices in each graph), aggregation (e.g., computing, for each pair of vertices, the variance of the shortest distance across graphs over time), filtering, and other operations. Thus, we require a framework that can conveniently and efficiently run complex analytic queries to “connect the dots” on collections of large graphs.

Existing systems do not effectively address the above challenges. For example, relational database systems require breaking down graph structures into edges recorded in a relation [52]. Therefore, graph analysis, using relational databases, involves costly join operations [16,52]. On the other hand, current graph processing systems such as Google’s Pregel [29], Giraph [4], GraphLab [28], GraphChi [24], Microsoft’s Trinity [48], the open source Neo4j [32], and others [10,11,14,19] can perform only one operation on one graph at a time. Thus, these systems cannot readily support complex queries on evolving graphs. 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 in Sect. 3.3 of this paper and experimentally demonstrated in Sect. 5.2.

We present a new parallel system, G^* , that conveniently and efficiently manages collections of large graphs by enabling scalable and distributed storage of graph data on multiple servers [41]. Each server is assigned a subset of vertices and their outgoing edges from multiple graphs for high data locality, which allows the server to access all of a vertex’s edges without contacting other servers. Since 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.

To quickly access disk-resident data about a vertex and its edges, each G^* server maintains an index that contains (*vertex ID*, *disk location*) pairs. This index also takes advantage of the commonalities among graphs to reduce its footprint. Specifically, this index stores only one (*vertex ID*, *disk location*) pair for each vertex version in a collection for the combination of graphs that contain that version. Due to its small size, the index can be kept fully or mostly in memory, enabling fast lookups and updates (Sect. 3.3). To prevent the graph index from managing too many combinations of graphs, each G^* server also automatically groups graphs and separately indexes each group of graphs (Sect. 3.4).

Like traditional database systems, G^* supports sophisticated queries using a dataflow approach where operators process data in parallel. In contrast to other database systems, G^* provides simple yet powerful processing primitives for solving graph problems. These primitives effectively hide the complexity of distributed data management and permit easy and succinct implementation of graph operators. Furthermore,

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 throughout this paper.

G* operators process a vertex and its edges once and then associate the result with all relevant graphs. This type of sharing accelerates queries on multiple graphs.

To the best of our knowledge, G* is the first system that provides all of the above features, in contrast with previous database and graph processing systems. Our contributions are as follows: We

- provide an architectural design for a parallel system that conveniently and quickly runs complex queries on collections of large graphs.
- develop techniques for efficiently storing and indexing large graphs on many servers.
- present a new parallel dataflow framework that can accelerate queries on multiple graphs by sharing computations across graphs.
- demonstrate the benefits of the above features with experimental results.

The rest of this paper is organized as follows: Sect. 2 provides an overview of the G* system. Sections 3 and 4 present G*'s storage and query processing frameworks, respectively. Section 5 shows evaluation results and Sect. 6 summarizes related work. Section 7 concludes this paper.

2 Background

This section describes G*'s system architecture (Sect. 2.1), data model (Sect. 2.2), and query languages (Sect. 2.3).

2.1 G* system architecture

As Fig. 1 shows, G* is a distributed system that consists of multiple servers. A server that manages the whole system is called the *master*. A query submitted to the master is first transformed by the *query parser* into a network of operators, which is then

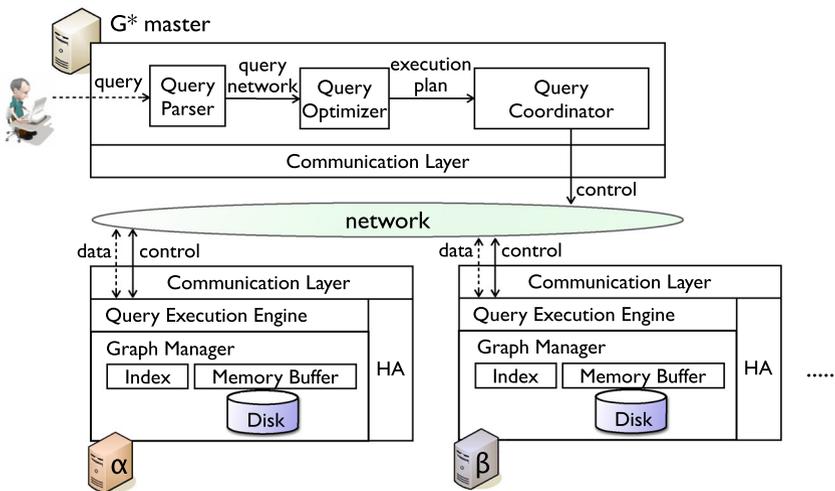


Fig. 1 G* architecture

```

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

```

Fig. 2 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)

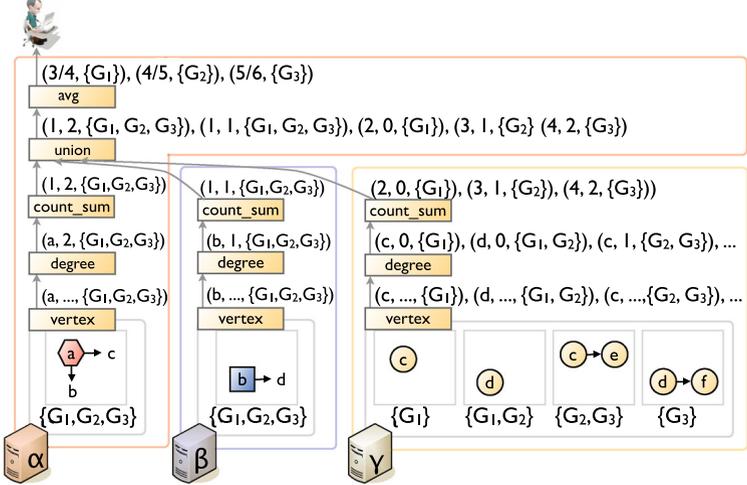


Fig. 3 Operator network for distributed computation of average degree

converted by the *query optimizer* into an optimized query execution plan (Fig. 2). Based on the execution plan, the *query coordinator* instantiates and executes operators on other servers (Fig. 3) 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. Finally, the *high availability* module performs tasks for masking server and network failures, which is not further discussed in this paper. The above components are currently implemented in approximately 30,000 lines of Java code.

2.2 Data model

2.2.1 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 joins required. For example, to retrieve all of the 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 a traversal increases. It has been shown that for the traversal-type queries, relational databases can be an order of magnitude or more slower than others based on non-relational technologies [50].

2.2.2 Our nested data model

To avoid the complications mentioned in Sect. 2.2.1, G* uses a *nested* data model to capture the inherent relationships among graphs, vertices, and edges in the following logical schema:

$$\text{graph}(\underline{\text{id}}, \text{att}_1, \text{att}_2, \dots, \{\text{vertex}\})$$

The absolute path to each graph on G*'s distributed file system is used as the primary key, $\underline{\text{id}}$. Each 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:

$$\text{vertex}(\underline{\text{id}}, \text{att}_1, \dots, \{\text{edge}\}),$$

where $\underline{\text{id}}$ is an identifier that distinguishes among the vertices in the same graph, each att_i is a vertex attribute, $\{\text{edge}\}$ is the set of edges emanating from the vertex identified by $\underline{\text{id}}$. Given multiple graphs and their vertices, the primary key for uniquely identifying a vertex is $\underline{\text{graph.id}}, \underline{\text{vertex.id}}$, where $\underline{\text{graph.id}}$ and $\underline{\text{vertex.id}}$ are the graph and vertex IDs mentioned above. For example, when $\underline{\text{graph.id}}$ is `'/twitter/1'` and $\underline{\text{vertex.id}}$ is `'a'`, vertex.address refers to the attribute named `address` of the vertex whose ID is `'a'` in a graph whose ID is `'/twitter/1'`.

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

$$\text{edge}(\underline{\text{id}}, \text{att}_1, \dots)$$

where $\underline{\text{id}}$ is an identifier that distinguishes among the edges based on the IDs of the vertices that the edges are incident to, and each att_i is an edge attribute. The primary key for uniquely identifying each edge is thus $\underline{\text{graph.id}}, \underline{\text{vertex.id}}, \underline{\text{edge.id}}$, where $\underline{\text{graph.id}}$, $\underline{\text{vertex.id}}$ and $\underline{\text{edge.id}}$ are the graph, vertex, and edge IDs mentioned above.

2.3 Query languages

G* currently supports two query language that are summarized in Sects. 2.3.1 and 2.3.2.

2.3.1 Procedural graph query language

The procedural graph query language (PGQL) can directly define a network of operators that the master server constructs on G^* 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 like `count`, `sum`, `min`, `max`, and `avg`. Other non-graph operators include `UnionOperator`, `ProjectionOperator`, `JoinOperator`, `SortOperator` and `TopKOperator`. For details and examples about all operators, please see our G^* Operator Reference [13].

The commands for creating operators have the following form:

```
<op_name> @ <server_id|* > = <op_type> ([ <input_op_names> ],
    <param_1> , <param_2> , ...)
```

Each operator creation command constructs a new `<op_type>`-type operator called `<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 parameter needed during operator creation. For example, in Fig. 2, `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 2 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 Fig. 2, all of the 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 Fig. 2) and execute the specified functions on that input based on the specified attributes and then stores the results in the specified output attributes.

Figure 3 illustrates a network of operators constructed according to the example in Fig. 2. Figure 3 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 α 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 Fig. 2). These partial aggregate values computed on each server are merged by the `union` operator on server α 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. Further details of query processing, including development of graph processing operators and shared computation across graphs, are discussed in Sect. 4.

2.3.2 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 [3] in that it enables queries defined upon sets of complex objects (e.g., vertices referencing edges to other vertices). We have an early version of a translator, built in ANTLR [35], that turns DGQL into PGQL.

Figure 4 shows example queries that are based on the representative applications mentioned in Sect. 1. These queries are also used in Sect. 5 to measure the performance of G^* . 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 Fig. 4, `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

```

1 select graph.id, avg(degree) -- Q1. average vertex degree
2 from (select graph.id, degree(vertex) as degree
3       from graph('/twitter/*'))
4 group by graph.id
5
6 select graph.id, coeff10*0.1, count(*) -- Q2. clustering coefficient dist.
7 from (select graph.id, floor(c_coeff(vertex)*10) as coeff10
8       from graph('/twitter/*'))
9 group by graph.id, coeff10
10
11 select graph.id, min_dist, count(*) -- Q3. minimum distance distribution
12 from min_dist(graph('/tree/*'), '1') -- vertices with min_dist
13 group by graph.id, min_dist
14
15 select graph.id, comp_size, count(*) -- Q4. component size distribution
16 from (select graph.id, comp_id, count(*) as comp_size
17       from comp_id(graph('/twitter/*')) -- vertices with comp_id
18       group by graph.id, comp_id)
19 group by graph.id, comp_size

```

Fig. 4 Dynamic graph queries—DGQL

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 Efficient storage of graphs

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 10 ms disk seek time, it would take 116 days to access 1 billion edges. This section presents a solution that meets these requirements.

3.1 Overview of graph data storage

G^* manages directed graphs 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 receive data from external sources such as Twitter's Garden hose [49] 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 (Sect. 3.3.3) and then updating only the newly created graph according to the new data.² G^* assigns a vertex and its outgoing edges to the same server for high data locality.³ For example, in Fig. 5a, server α can access every edge of vertex a without contacting others. An update of an edge is therefore handled by the server that stores the source vertex of that edge.

Each G^* server strives to efficiently manage data by taking advantage of commonalities among the graphs. For example, server α in Fig. 5a is assigned vertex a and its outgoing edges which remain the same in graphs G_1 , G_2 and G_3 . Thus, server α 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 (Fig. 5b). In response to this update, server γ 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. Section 3.2 discusses the details of efficiently storing these versions.

² In this paper, we focus on managing graphs that correspond to periodic snapshots of an evolving network. Logging the input data allows G^* to reconstruct graphs as of any points in the past by using periodic snapshots and log data. This feature is not further discussed in this paper.

³ The current G^* implementation assigns each vertex to a server based on the hash value of the vertex ID. We are developing data distribution techniques that can reduce the edges whose end points are assigned to different servers.

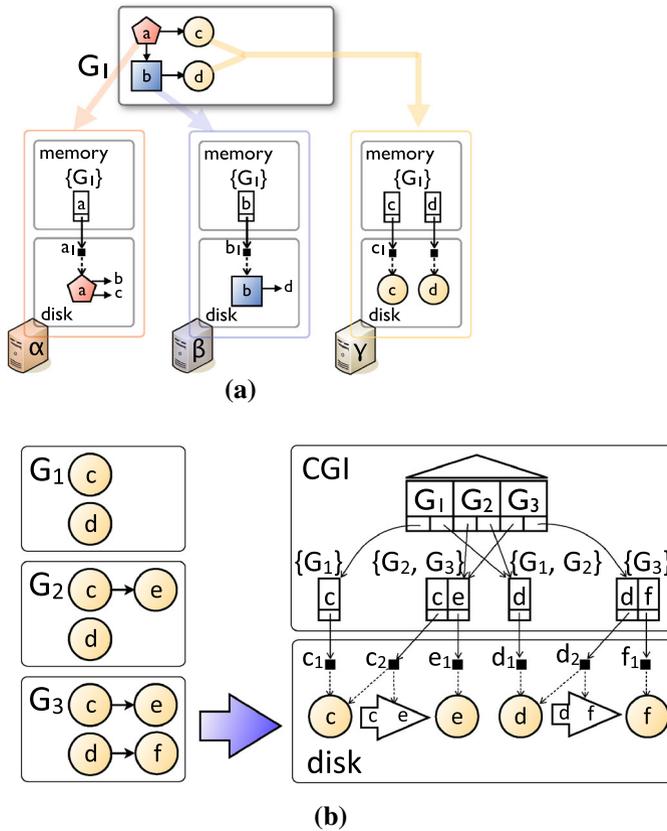


Fig. 5 Overview of G^* : efficient storage and indexing for sequential graph snapshots. **a** Initial creation of graph G_1 at time 1, when each vertex and its edges are assigned to a server. In this example, servers α and β are unchanged at times 2 and 3, **b** Server γ at time 3. Vertex e and edge (c, e) having been added in graph G_2 at time 2, server γ 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 γ 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)

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 index the *Compact Graph Index* (CGI). Specifically, this index stores only one (*vertex ID, disk location*) pair for each vertex version in a collection for the combination of graphs that contain that version. We call this collection a *VL map* (“Vertex Location” map) since it associates a vertex ID with a disk location. We call a (*vertex ID, disk location*) pair a *VL pair*. In Fig. 5b, vertex version c_2 on server γ represents vertex c and its outgoing edges which remain the same in graphs G_2 and G_3 . For c_2 , γ ’s CGI stores $(c, \text{location}(c_2))$ 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,

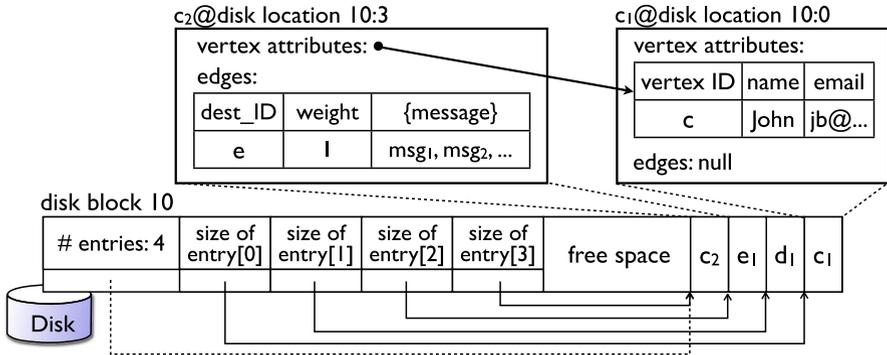


Fig. 6 Organization of a disk block. Objects are allocated from the end of the block while data about these objects is stored from the beginning of the block

enabling fast lookups and updates (Sect. 3.3). 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 (Sect. 3.4).

3.2 Disk storage of graph data

In a variety of graph applications, the edges of a vertex must be processed together (Sect. 4). 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 256 KB in the current implementation.) For each vertex, the vertex ID, attribute values, and all of the outgoing edges are stored on disk (Fig. 6). For each edge, the ID of the destination vertex and the attribute values of the edge are saved on disk. In Fig. 6, two versions c₁ and c₂ of vertex c are stored within disk block 10 at indices 0 and 3, respectively. For space efficiency, c₁ and c₂ 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 pool* that keeps a memory cache of disk blocks (Fig. 1).

3.3 Compact graph index

As Fig. 7 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 KB 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

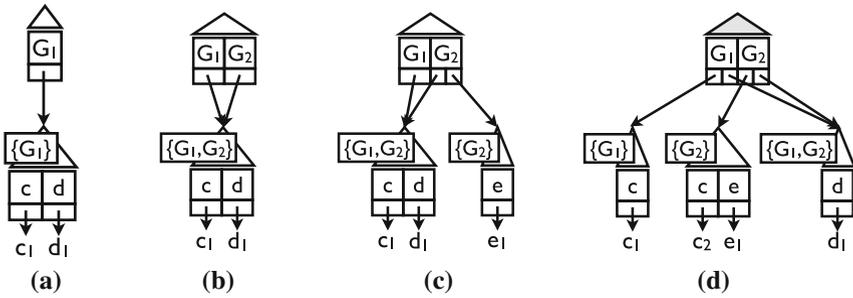


Fig. 7 CGI update examples (a shows server γ 's CGI in Fig. 5a). (a) Adding G_1 , (b) G_2 as a clone of G_1 , (c) addition of e in G_2 , (d) update of vertex c in G_2

Table 1 Symbols for CGI cost analysis

Symbol	Description
N	Number of graphs (graphs: G_1, G_2, \dots, G_N)
$N(v)$	Number of graphs in $\{G_i\}_{i=1}^N$ that contain vertex v
$N(v, Q)$	Number of graphs, among those in Q , that contain v
V	Set of all distinct vertex IDs in $\{G_i\}_{i=1}^N$ (i.e., size of the set)
$M(v)$	Number of VL maps that contain vertex v (i.e., number of versions of v)
$M(v, Q)$	Number of VL maps that contain v and are related to a graph in Q (i.e., number of distinct versions of v in graphs Q)
$M(G_i)$	Number of VL maps related to graph G_i
$s(G_i)$	The size of the ID of graph G_i
$p(v)$	The size of VL pair for vertex v

each graph ID with all of the relevant VL maps (see the shaded triangle in Fig. 7d 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 (Sects. 3.3.1, 3.3.3). As experimentally demonstrated in Sect. 5.2, 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_α and remains the same in the subsequent graphs until it is superseded by a new version in graph G_ω . This means that common vertices and edges always belong to graph combinations of the form $\{G_i\}_{i=\alpha}^{\omega-1}$. Section 3.4 presents a technique for controlling the overhead of the CGI. This technique groups graphs and constructs a separate CGI for each group.

3.3.1 Space cost analysis

Table 1 shows the symbols used in the CGI cost analysis. The results are summarized in Table 2 and experimentally verified in Sect. 5.

Table 2 Graph indexing cost analysis: CGI and PGI

	CGI	PGI
Space	$\sum_{v \in V} M(v)p(v) + \sum_{i=1}^N M(G_i)s(G_i)$	$\sum_{v \in V} N(v)p(v) + \sum_{i=1}^N s(G_i)$
put(<i>v</i> , <i>d</i> , <i>g</i>)	$O(N + \log V)$	$O(\log V)$
clone(<i>g'</i> , <i>g</i>)	$O(NM(g))$	$O(V)$
getLocation(<i>v</i> , <i>Q</i>)		
Index	$O(M(v)N + M(v, Q) \log V)$	$O(Q \log V)$
Data	$O(M(v, Q))$	$O(N(v, Q))$

Given vertex *v* and each version *v_i* of *v*, the CGI contains only one VL pair in the VL map for the graphs that contain *v_i*. Therefore, the total amount of space for storing all of the VL pairs can be expressed as $\sum_{v \in V} M(v)p(v)$ where *V*, *M(v)* and *p(v)* are as defined in Table 1. The space overhead expression for the CGI in Table 2 includes an additional term, $\sum_{i=1}^N M(G_i)s(G_i)$, to account for the space used for the graph IDs contained in the CGI.

In contrast with the CGI, consider a naive index structure that maintains a separate VL map for each graph, which is called the *Per-Graph Index* (PGI) in Table 2. In the PGI, the space required for storing all of the VL pairs is $\sum_{v \in V} N(v)p(v)$ since the PGI contains one VL pair for every graph that contains *v*. In this case, the amount of space required for storing graph IDs is $\sum_{i=1}^N s(G_i)$.

The above analysis shows that the CGI becomes more compact than the PGI as each vertex has fewer versions and membership in more graphs. For example, if there were 100 graphs and 5 distinct versions of vertex *v* in these graphs, the CGI will contain only 5 VL pairs for *v* whereas the PGI will contain 100 pairs for *v*. In the unlikely situation where every vertex and its edges are updated in each graph (perhaps, if every Twitter user sends a message to a new user every hour), the CGI converges to the PGI because there is no commonality among graph versions and therefore one VL map is required for each vertex in each graph.

3.3.2 Compact graph index creation and updates

The CGI starts with an empty root map. When the first graph *G₁* is registered into the CGI, the root map adds an entry that contains *G₁*'s ID. *G₁*'s vertices and edges are saved to disk as explained in Sect. 3.2 while only the vertex IDs and disk locations are inserted into the CGI. Since all or most of the CGI can be kept in memory, the cost of this update is negligible compared to the overhead of storing the graph data on disk.

The put(VID *v*, DLOC *d*, GID *g*) method in Figure 8 relates the specified vertex ID (VID) *v*, disk location (DLOC) *d* of the vertex data, and graph ID (GID) *g* to each other. This sequence of events is illustrated in Fig. 7.

The method first determines whether or not vertex *v* is already contained in an existing VL map (lines 2–3). If there is no such VL map (line 4), a VL map which is related only to the target graph *g* is found by using getVLMap(*g*) (line 5). getVLMap(*g*) creates a new VL map if none existed before. Then, (*v*, *d*) is stored in this VL map

```

1 public void put(VID v, DLOC d, GID g) {
2     VLMap m1 = getVLMap(v, d); // VL map having (v, d)
3     VLMap m2 = getVLMap(v, g); // VL map related with g, having v
4     if (m1 == null && m2 == null) // if no VL map contains v
5         getVLMap(g).put(v, d); // put (v, d) in VL map only for g
6     if (m1 != null && !m1.related(g)) { // m1 not related with g
7         m1.removeVertex(v);
8         common(m1, g).put(v, d);
9     }
10    if (m2 != null) { // if m2 related with g, contains v
11        DLOC prevD = m2.location(v); // previous disk loc. of v
12        if (!d.equals(prevD)) { // current disk loc. of v differs
13            m2.removeVertex(v);
14            common(m1, g).put(v, d);
15            if (!m2.isOnlyRelated(g)) // m2 not related only with g
16                diff(m2, g).put(v, prevD);
17        }
18    }
19 }

```

Fig. 8 CGI update method

(line 5). For example, in Fig. 7b, information about vertex e is not contained in any VL map. Therefore, when vertex e is added in graph G_2 , the VL map for $\{G_2\}$ stores e 's ID and disk location (Fig. 7c).

On the other hand, if the (v, d) pair is contained in a previous VL map m_1 (lines 2 and 6 in Fig. 8), the method determines whether or not m_1 is related to the target graph g , using $m_1.related(g)$ on line 6. If so (i.e., (v, d) is already contained in the VL map related to g), no further action is needed. Otherwise, (v, d) must be moved from m_1 (line 7) to a VL map that is related to graph g as well as all the graphs related to m_1 . This map is found using $common(m_1, g)$ on line 8.

Suppose that a vertex v in graph g is updated. In this case, the new version of the vertex is stored at a different disk location d rather than the previous disk location $prevD$ which preserves v 's previous version. Next, a VL map m_2 , which is related to graph g and contains information about v , is found (lines 3 and 10). For example, in Fig. 7c, if c is updated in graph G_2 , the VL map for $\{G_1, G_2\}$ corresponds to m_2 since it contains information about c . Then, $(v, prevD)$ is removed from m_2 (line 13) and stored in the VL map related to all of the graphs that contain $(v, prevD)$, but not graph g . The VL map needed to store $(v, prevD)$ can be found by $diff(m_2, g)$ on line 16. For this reason, in Fig. 7c, d, data about c_1 (i.e., the previous version of c) is moved from the VL map for $\{G_1, G_2\}$ to the VL map for $\{G_1\}$. Furthermore, (v, d) needs to be stored in the VL map related to graph g as well as all the other graphs that contain that version of v at disk location d . That VL map is found using $common(m_1, g)$ on line 14. In Fig. 7c, d, information about the new version, c_2 , of c is stored in the VL map for $\{G_2\}$.

Cost analysis The $put(v, d, g)$ method runs in $O(N + \log V)$ time. This is because all of the VL maps that contain vertex v (lines 2–3 in Fig. 8) can be found in $O(M(v))$ time using an inverted list for v ($maps(v)$) that points to the VL maps which contain v . Given a list of graph IDs, the VL map related to all of the corresponding graphs (lines 5, 8, 14, 16) can be found in $O(N)$ time using a hash-map that associates

a sorted list of graph IDs with each relevant VL map. Other operations that insert (or remove) a VL pair into (or out of) a VL map (lines 5, 7, 13, 14, 16) and that find the disk location of v (line 11) can be completed in $O(\log V)$ time since a B+ tree is used for each VL map. Also, $M(v) \leq N$, therefore, all of the above operations can be completed in $O(N + \log V)$ time.

In contrast to the CGI, the PGI can complete $\text{put}(v, d, g)$ in $O(\log V)$ time by inserting the (v, d) pair into the VL map for g . This indicates that the CGI has a relatively higher update cost as more graphs are indexed together. However, this extra overhead is negligible in practice when compared to the cost of writing data to disk. Furthermore, Sect. 3.4 presents a solution that can trade space for a faster update speed.

3.3.3 Graph cloning

We can obtain a series of cumulative graphs by iteratively cloning the last graph and then adding vertices and edges to the new graph. As Fig. 7a, b show, the CGI can quickly create a new clone g' of graph g by updating the root map and the VL maps related to g so that they are also related to g' .

Cost analysis The clone operation can be completed in $O(NM(g))$ time since $M(g)$ VL maps are related to g and each of these VL maps needs to be related to g' . Associating a VL map with g' takes $O(N)$ time since it requires updating the hash-map that associates a sorted list of graph IDs with relevant VL maps (explained in Sect. 3.3.2). In contrast to the CGI, the PGI takes the substantially longer $O(V)$ time because it must replicate the entirety of the VL map for g .

3.3.4 Graph data retrieval

To efficiently process queries on multiple graphs, the CGI supports the $\text{getLocations}(v, Q)$ method in Fig. 9. Given a vertex ID (VID) v and a set of graph IDs ($\text{Set}\langle\text{GID}\rangle$) Q , the method returns a collection of pairs representing the disk location that stores a version of v in Q and the IDs of all of the graphs which contain that version. For example, in Fig. 7d, if the disk locations in Fig. 6 are assumed, a call to getLocations passing in vertex d and graphs G_1 and G_2 (i.e., $\text{getLocations}(d, G_1, G_2)$) would return $\{(10:1, \{G_1, G_2\})\}$

```

1 Set<Pair<DLLOC, Set<GID>>> getLocations(VID v, Set<GID> Q) {
2   Set<Pair<DLLOC, Set<GID>>> locations = new HashSet();
3   for (VLMMap m : maps(v)) { // for each VLMMap having v
4     Set<GID> R = m.relatedGraphs(Q);
5     // graphs among those in Q that are related with m
6     if (R.size() > 0)
7       locations.add(new Pair(m.location(v), R));
8   }
9   return locations;
10 }
```

Fig. 9 CGI lookup method

and `getLocations(c, {G1, G2})` would return $\{(10:0, \{G_1\}), (10:3, \{G_2\})\}$. This `getLocations(v, Q)` method allows G* operators to process each vertex version once and then use the result across all of the graphs which contain that vertex version (Sect. 4.2). This substantially speeds up queries on multiple graphs (Sect. 5.1).

Cost analysis The `getLocations(v, Q)` method iterates over the VL maps which contain information about vertex v , using the inverted list for v (line 3 in Fig. 9). This inverted list, `maps(v)`, which contains $M(v)$ VM maps is explained in the cost analysis of the `put(v, d, g)` method. The `getLocations(v, Q)` method then finds, for each VL map m in `maps(v)`, the set of graphs R that are contained in Q and related to m (line 4). Finding R takes $O(N)$ time since it requires finding the intersection of two sorted lists of graph IDs. This set R of graphs, if it is non-empty, is added to the result set in conjunction with the disk location that stores the state of vertex v when v belongs to the graphs in R (lines 6–7). The disk location of v can be found in $O(\log V)$ time using the B+ tree for VL map m . Therefore, the overall time complexity of the `getLocations(v, Q)` method is $O(M(v)N + M(v, Q) \log V)$ where $M(v, Q)$ denotes the number of VL maps that contain information about vertex v and that are related to a graph in Q (i.e., the number of distinct versions of v in graphs Q). Reading the $M(v, Q)$ versions of v from disk via the buffer pool may require up to $M(v, Q)$ disk accesses (Table 2) since we never need more than one disk access to read each version of v . On the other hand, accessing a vertex once for each graph using the PGI would cause up to $G(v, Q)$ disk accesses, where $G(v, Q)$ denotes the number of graphs, among those in Q , that contain v .

Example Suppose that there are 100 graphs and 5 versions of vertex v in these graphs. Suppose further that the CGI is kept in memory due to its small size. In this case, a query on these graphs requires up to 5 disk accesses for vertex v , one for each version of v . If the PGI were used, the same query would require up to 100 disk accesses for vertex v , one for each graph. In this example, the CGI would be much faster than the PGI.

3.4 Compact graph index splitting

The CGI has low space overhead, enables the sharing of computations across graphs, and may substantially reduce the data retrieval overhead. However, as more graphs are added to the CGI, the number of VL maps may grow superlinearly, thereby noticeably increasing lookup and update overhead. Our solution to this problem groups graphs and then constructs a separate CGI for each group of graphs in order to limit the number of VL maps managed by each CGI. In other words, this approach trades space for speed by ignoring the commonalities among graphs in different groups. Figure 10 shows an example where one CGI is constructed for every three graphs. The root collection in the example associates each graph (e.g., G_1) with the CGI which covers that graph (e.g., the CGI for G_1, G_2 and G_3).

One important challenge in implementing this approach is to effectively determine the number of graphs to be covered by each CGI. Figure 11 shows our solution to determine whether or not to split the current CGI into two: one that covers the previous graphs and another one to cover the current and succeeding graphs. The code in Fig. 11

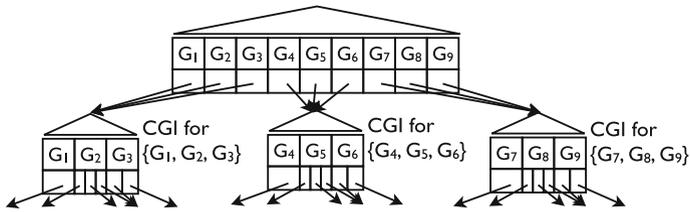


Fig. 10 CGI splitting example

```

1 boolean needSplit() {
2     return (lookupDelays.size() >= windowSize &&
3         curDelay() - minDelay > threshold*minDelay);
4 }

```

Fig. 11 CGI splitting code

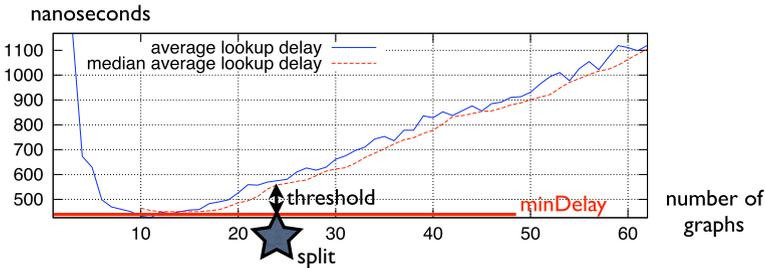


Fig. 12 Determination of split point. The lookup delay initially decreases because each lookup returns data relevant to more graphs until too many VL maps are created, causing the lookup delay to increase

is invoked whenever a new graph is added. This method first determines if sufficient average lookup delays have been entered into a list `lookupDelays` (line 2).⁴ If so (i.e., the size of the `lookupDelays` list is larger than a threshold `windowSize`), the method determines whether or not there has been a substantial increase in the lookup delay when compared to the minimum delay observed in the past (line 3). To compute the average lookup delay for each graph, each CGI maintains two variables that keep track of the sum and count of lookup delays for each graph.

To capture the general trend in the lookup delay despite measurement inaccuracies (see Fig. 12), our approach uses a sliding window on the series of average lookup delays while selecting the median whenever the window advances in response to the addition of a new graph. `curDelay` on line 3 refers to the median delay selected from the current window (the dotted line in Fig. 12) and `minDelay` refers to the minimum among all of the previous median delays. In the current implementation, `threshold` is set to 0.1 to detect any 10 % increase in the lookup delay and `windowSize` is set to 10 by default. Experimental results in Sect. 5.2 will show that our CGI

⁴ As mentioned in the cost analysis of the `put(v, d, g)` method, updating the CGI for a version of vertex `v` also requires a lookup via `maps(v)`.

splitting approach makes reliable decisions while keeping the lookup delay near the minimum.

4 Graph query processing

Traditional database systems transform queries into a network of operators that process data in a pipelined fashion. A central challenge in applying this approach to G^* is to develop a new query processing framework that meets the following requirements:

1. The query processing framework must be able to execute operators on multiple servers to efficiently process distributed graphs.
2. G^* 's indexing mechanism associates each vertex with all of the graphs that contain that vertex (Sect. 3.3). The query processing framework must use this feature to share computations across relevant graphs.
3. G^* requires operators for solving graph problems. The framework must permit easy and succinct implementation of these operators.

Sections 4.1, 4.2 and 4.3 describe how we address the above issues.

4.1 Query processing framework

Given a query plan (Fig. 2), the G^* master server constructs a network of operators (Fig. 3) according to that plan. Each G^* server, including the master, interacts with the others using remote method invocations (RMIs) [17]. For example, given the command

```
vertex@* = VertexOperator([], '/twitter/*')
```

(line 1 in Fig. 2), the master invokes the `createOperator()` method on each G^* server while passing in (1) the label to assign to that operator (`vertex`), (2) the type of the operator (`VertexOperator`), (3) the operators to connect to for input data (`[]`, 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 Fig. 3, can obtain data from another operator by receiving an *iterator* from that operator and then repeatedly calling `next()` on that iterator. Just like traditional database management systems, this iterator-based approach is for pipelined transmission and processing of data. Since the native Java RMI [17] does not directly support methods that return an iterator, we constructed our own RMI framework to overcome this limitation. Furthermore, G^* utilizes Java externalization to speed up data storage and transmission (i.e., whenever possible, it writes and reads primitive data values using custom code instead of incurring the higher overhead of serializing/deserializing objects).

Using our RMI service, the `union` operator on server α in Fig. 3 obtains an iterator for getting data from the remote operator `count_sum` on server β . In this case, the `union` operator is given a proxy iterator constructed on server α 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, aggregation, and join. These operators may directly read disk-resident graph data (e.g., the `vertex` operator in Fig. 3), 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 (Sect. 4.3). 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).

4.2 Sharing of computations across graphs

Each `vertex` operator in Fig. 3 obtains an iterator from the set of CGIs (Sect. 3.4) that cover the graphs being queried (e.g., $\{G_1, G_2, G_3\}$). 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., $(a, \dots, \{G_1, G_2, G_3\})$ on server α in Fig. 3). In G^* , the `Vertex` type is used for these objects. In summary, if a vertex and its edges do not change across multiple graphs, the `vertex` operator loads them only once from disk and then associate 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 Fig. 3 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 G_1 , G_2 , and G_3 . 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 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. The utility of this shared computation is experimentally demonstrated in Sect. 5.

```

1 public Vertex next() {
2     Vertex v = input(0).next();
3     v.add("degree", v.degree());
4     return v;
5 }

```

Fig. 13 Vertex degree computation

Figure 13 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.

4.3 Primitives for graph processing

To facilitate the implementation of graph processing operators, G^* provides three types of primitives: *summaries*, *combiners*, and bulk synchronous parallel (*BSP*) operators.

4.3.1 Summaries

Graph algorithms typically maintain certain types of values (e.g., the shortest distance from a chosen vertex) for each vertex [10, 14, 29]. A *summary* is a container that keeps aggregate data (e.g., *count*, *sum*) to support operators for solving graph problems. One can implement a custom summary type by implementing the `Summary<V, F>` interface, where V is the type of the values for updating aggregate data, and F is the type of the value to generate from the aggregate data. Each summary type must implement the following methods:

- `boolean update(V v)`: updates the aggregate data using value v (e.g., `sum += v; count++`;) and then returns `true` if the aggregate data is changed; `false` otherwise.
- `boolean update(Summary<V, F> s)`: updates the aggregate data using other summary s (e.g., `sum += s.sum; count += s.count`;) and returns `true` if the aggregate data is changed; `false` otherwise.
- `F value()`: returns a value computed using the current aggregate data (e.g., `return sum/count`;).

Figure 14 shows a portion of the actual implementation of a summary which computes the clustering coefficient of a vertex. Clustering coefficients can be used to determine whether or not a given graph represents a small-world network [22]. The

```

1 public boolean update(Vertex v) {
2     boolean updated = false;
3     for(Vertex.Edge e: v.edges())
4         if (neighbors.contains(e.target())) {
5             triangles++;
6             updated = true;
7         }
8     return updated;
9 }
10
11 public Double value() {
12     return 1.0*triangles/neighbors.size()/(neighbors.size()-1);
13 }

```

Fig. 14 Implementation of `CCoeffSummary`

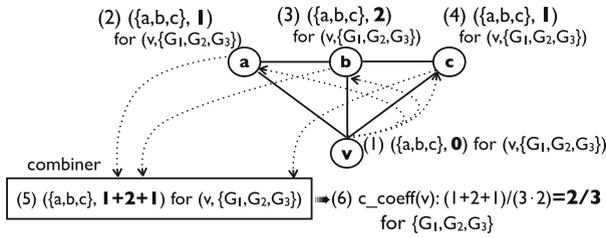


Fig. 15 Clustering coefficient example

clustering coefficient of a vertex v is defined as the ratio of the number of edges between neighbors of the vertex (i.e., the number of *triangles* that involve v) to the maximum number of edges that could exist between neighbors of v . For instance, vertex v in Fig. 15 has three neighbors (a, b, c). Vertex a has one edge to a neighbor of v (b). Vertex b has two such edges ($b-a$ and $b-c$). Vertex c has one such edge ($c-b$). Therefore, the clustering coefficient of vertex v is $(1 + 2 + 1)/(3 \cdot 2) = 4/6 = 2/3$.

The summary `CCoeffSummary` in Fig. 14 has two variables: (i) `neighbors`, a collection that contains the IDs of the neighbors of a vertex, and (ii) `triangles`, an `int` variable for counting the number of edges between the neighbors.

Figure 15 shows an example where the clustering coefficient of vertex v is computed using three `CCoeffSummary` instances whose `neighbors` and `triangles` are initialized in step (1) to $\{a, b, c\}$ and 0, respectively. In step (2), a summary is updated based on vertex a using the `update(Vertex v)` method in Fig. 14. In this case, `triangles` is incremented once due to the edge from a to b , which is another neighbor of v . Two other summaries are updated similarly in steps (3) and (4). In step (5), these summaries are combined using the `update(Summary <Vertex, Double> s)` method (omitted in Fig. 14). In step (6), the clustering coefficient of v is computed using the `value()` method in Fig. 14.

4.3.2 Combiner

The example in Fig. 15 updates summaries based on vertices. It also combines all of the summaries that have the same target vertex (i.e., the vertex for which the clustering coefficient is computed) to compute the final value. Our `combiner` primitive allows operators to perform the above operations while ignoring the low-level details of distributed data management. It also abstracts away the complexity of maintaining summaries for collections of graphs that are queried together. The `combiner` provides the following methods:

- `void update(VID t, Summary <V,F> s, Set <GID>g)`: associates summary s with vertex t and a set of graphs g . If a summary is already associated with t , that summary is updated using s .
- `void update(VID t, Summary <V,F> s, VID i, Set <GID>g)`: updates summary s based on vertex i and then performs `update(t, s, g)`.
- `boolean hasNext()`: returns `true` if the combiner can compute a new final value (e.g., the clustering coefficient of a vertex) since it has received all of the needed summaries. This method returns `false` if no further new values can be

```

1 public void init() {
2     while(input().hasNext()) {
3         Vertex v = input().next();
4         for (VID n : v.neighbors())
5             cmb.r.update(v.id(), new CCoeffSummary(v.neighbors()), n, v.graphs());
6     }
7 }
8
9 public Vertex next() {
10     return cmb.r.next();
11 }

```

Fig. 16 Clustering coefficient computation

computed (i.e., the summaries for all of the vertices are completely processed). This method blocks if the above is not yet known.

- `Vertex next()`: returns a `Vertex` object after inserting a new final value into that object (e.g., inserting the clustering coefficient of v into an object representing v). This method blocks in situations where `hasNext()` blocks.

Figure 16 shows the actual implementation of an operator that computes the clustering coefficient of each vertex. As the initial task, the operator processes a stream of `Vertex` objects (lines 2–6). For each `Vertex` object v (line 3), the operator uses a combiner `cmb.r` to route a summary to each neighbor n of v as in step (1) in Fig. 15 (lines 4–5 in Fig. 16). In this case, the underlying G^* server sends that summary to the server that manages vertex n . When summaries return from their trip to a neighbor of v , the G^* server combines these summaries using `update(Summary < V, F > s)`. By calling `value()` on the resulting summary, the server computes the clustering coefficient of v and adds it to the `Vertex` object that represents v . Whenever a `Vertex` object becomes available, the `next()` method of `cmb.r` returns the object (lines 9–11 in Fig. 16).

4.3.3 BSP operator

The BSP model has been frequently used in various parallel graph algorithm implementations [11, 14, 29]. 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 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 implementing 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.

```

1 void init() {
2   cmbr.update(src, new Min<Double>(0.0), graphs(graphPred));
3 }
4
5 void compute(Vertex v, Summary<Double, Double> s) {
6   for (Edge e : v.edges()) {
7     cmbr.update(e.target(), new Min<Double>(s.value() + e.weight()), v.graphs
8       ());
9   }
10 }

```

Fig. 17 Shortest distance computation

```

1 protected void init() {
2   Iterator<Vertex> i = vertices(graphPred);
3   while (i.hasNext()) {
4     Vertex v = i.next();
5     compute(v, new Min<VID>(v.id()));
6   }
7 }
8
9 protected void compute(Vertex v, Summary<VID, VID> s) {
10  for (Edge e : v.edges(edgePredicate))
11    cmbr.update(e.target(), s, v.graphs());
12 }

```

Fig. 18 Connected components computation

Figure 17 shows an operator implementation that finds, in each graph that satisfies predicate `graphPred`, the shortest distance from a single source vertex to every other vertex. The presented code is equivalent to the Pregel counterpart by Malewicz et al. [29]. In Fig. 17, a summary `Min<V>` is used to keep track of the shortest distance to each vertex.

In Fig. 17, the `init()` method assigns a summary containing 0.0 to the source vertex (`src`). Since this summary is newly associated with `src`, the `compute()` method is invoked on the `src` object during the next superstep. For each edge `e` of the current vertex `v` (line 6), this method computes the distance to the neighbor vertex (`e.target()`) and sends the distance in the form of a `Min<Double>` summary (line 7), which will later be combined with all of the other summaries sent to the same vertex. The `compute()` method is invoked on a vertex only when a shorter distance to the vertex is found in the previous superstep. If no summaries in the system are updated (i.e., no shorter distance is found for any of the vertices), the whole BSP process completes. In this case, `cmbr.next()` returns one `Vertex` object at a time, which contains the shortest distance from the source vertex.

Figure 18 shows another operator implementation that finds all of the connected components in each undirected graph that matches predicate `graphPred`. Initially, every relevant vertex in the system is assigned a `Min<Double>` summary which contains the ID of the vertex (lines 1–7). In each superstep, the `compute()` method is invoked only on the vertices to which a smaller ID value is sent. These vertices send the received ID to their neighbor vertices (lines 9–12). In this way, the smallest vertex ID within a connected component is eventually sent to all of the vertices within the component. This smallest ID is used as the component ID. The rest of this implemen-

tation which is omitted in Fig. 18 is the same as that of the shortest distance operator (Fig. 17).

4.3.4 Discussion

G*'s processing primitives have the following benefits over others' primitives [10, 14, 29]:

1. Summary implementations promote code reuse (e.g., the $\text{Min} < V >$ summary is used in Figs. 17, 18).
2. Summaries allow implementation of graph algorithms at a high level (e.g., simply use $\text{Min} < V >$ rather than writing code that finds the minimum among many values). For this reason, summary-based operator implementations are succinct. For example, the `compute()` method in Fig. 17 has only 5 lines of code whereas the equivalent code in Pregel [29] is 12 lines.
3. Summaries with the same target vertex are combined into one as they arrive at the destination server. In contrast, Pregel-like systems incur high space overhead since they keep all of the input messages until they are consumed by the user-defined code [10, 14, 29].
4. G*'s processing primitives effectively hide the complexity of dealing with multiple graphs. In the 50 lines of code from Figs. 13, 14, 16, 17 and 18, only line 5 in Fig. 16, line 7 in Fig. 17, and line 11 in Fig. 18 reveal the presence of multiple graphs.

5 Experimental results

This section presents experimental results that we have obtained by running G* on a 64-core server cluster. In this cluster, each of eight nodes has two Quad-Core Xeon E5430 2.67 GHz CPUs, 16GB RAM and a 2TB hard drive.

Table 3 summarizes the datasets used for the experiments. The *Twitter* dataset contains a subset of messages sent between Twitter users during February 2012. This dataset was collected using Twitter's Garden hose API [49], which supplied a 10 % sample of all tweets. In our sample of the data, each Twitter message was typically delivered to one through three Twitter users. We suspect that these users would correspond to 10–30 users in the actual Twitter system. The *Yahoo!* dataset contains communication records between end-users in the Internet and Yahoo! servers [51]. The *Tree* dataset was obtained by running a binary tree generator. Using each of the *Twitter* and *Yahoo!* data sets, we constructed a series of graphs that are hourly snapshots of the underlying network. A graph in each series was constructed by first cloning the previous graph (Sect. 3.3.3) and then adding new vertices and edges to

Table 3 Datasets

	Twitter	Yahoo!	Tree
# Of vertices	33 M	112 M	1 B
# Of edges	62 M	335 M	1 B
# Of records	78 M	1.1 B	1 B
Data size	11 GB	107.9 GB	21 GB

```

1 select graph, src,
2   1.0*count(i)/count(distinct des)/(count(distinct des)-1)
3 from edge natural left outer join (
4   select edge.graph, edge.src, e.src i, e.des
5   from edge join edge e
6   on edge.graph = e.graph and edge.des = e.src
7 ) t
8 group by graph, src
9 having count(distinct des) > 1
    
```

Fig. 19 Clustering coefficient in SQL. Lines 4–6 find all of the 2-hop paths from *src* to *des* via *i*. The left outer join on lines 3–7 then finds all of the edges between the neighbor vertices *i* and *des* of vertex *src* while keeping all of the edges from *src*. Line 1 computes the ratio of the number of edges between the neighbors of *src* (i.e., $\text{count}(i)$) to the maximum number of edges that could exist between the neighbors (i.e., $\text{count}(\text{distinct des})/(\text{count}(\text{distinct des})-1)$)

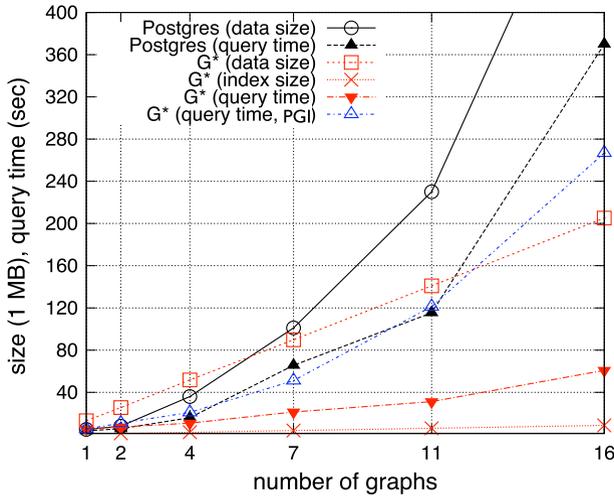


Fig. 20 G* versus PostgreSQL

the new graph. Our data sets did not contain records corresponding to the deletion of vertices and edges or edge weight updates. On these graphs, we ran the queries discussed in Sect. 2.3 (Fig. 4).

5.1 Comparison with prior database and graph systems

5.1.1 G* versus PostgreSQL

Figure 20 shows results that highlight the superiority of G* over PostgreSQL [37], a widely used relational database system. These results were obtained by running PostgreSQL and G* on identical servers. We used a set of 16 cumulative graphs constructed from the Twitter dataset. Each graph contained 3,000 additional edges compared to its previous graph. (Thus, the last graph contained 48,000 edges.) We kept the number of edges relatively small due to the performance limitations of PostgreSQL. PostgreSQL used a table `edge(graph, src, des)` to store data about the edges

in the graphs. We computed the clustering coefficient of every vertex in these graphs with the SQL query in Fig. 19.

Figure 20 shows that PostgreSQL has much higher storage overhead than G* (compare “Postgres (data size)” and “G* (data size)”). The reason is that Postgres stores one record for *every* combination of an edge and a graph, whereas G* stores each vertex and its edges only once regardless of the number of graphs that contain them.

In our measurement, the PostgreSQL data size and index size were nearly identical because the index covered all attributes (i.e., `graph`, `src`, `des`) of the primary key. By comparing “Postgres (data size)”, which also represents the index size of Postgres, with “G* (index size)”, we can see that the index size of G* is also much smaller than that of Postgres. The reason for this is that G*'s index contains (*vertex ID*, *disk location*) pairs in a deduplicated fashion.

In G*, the index size is in general much smaller than the amount of disk-resident graph data (compare “G* (index size)” with “G* (data size)”). Therefore, the entirety or a large fraction of the index can be kept in memory, enabling fast data lookup. Furthermore, G* processes each vertex once and then shares the result across relevant graphs. The curves labeled “Postgres (query time)” and “G* (query time)” in Fig. 20 clearly show the performance benefit of G* over PostgreSQL.

To study the importance of sharing computations across graphs, we examined another situation where G* constructed one VL map for each graph, ignoring commonalities among graphs (see “G* (query time, PGI)”). Figure 20 shows that “G* (query time)” and “G* (query time, PGI)” are significantly different, whereas “G* (query time, PGI)” and “Postgres (query time)” are similar. This result shows that G*'s ability to share computations across graphs is a main contributor to the superiority of G* over PostgreSQL.

5.1.2 G* versus Phoebus

The next result compares G* with Phoebus [36], an open-source implementation of Pregel [29]. We used Phoebus because the original Pregel system was not publicly available. As with previous work on Pregel [29], we performed single-source shortest distance queries on complete binary trees. One major difference is that we ran queries on multiple trees (i.e., graphs) rather than a single a tree. This experiment used a series of 10 cumulative graphs, each of which contained 25,000 more edges than its predecessor in the series.

Figure 21 shows the result obtained by running Phoebus and G* on identical servers (refer to Sect. 5.3 for our results on the scalability of G*). In this result, “Phoebus (last graph)” and “G* (last graph)” represent the amount of time that each system took to process the largest graph. This result shows that G* substantially outperforms Phoebus even when it processes a single graph. “Phoebus (all graphs)” and “G* (all graphs)” demonstrate that the performance difference between Phoebus and G* becomes larger when they process multiple graphs since G* can share computations across graphs, whereas Phoebus does not. When we disabled shared computation by using the PGI (Sect. 3.3.1) instead of the CGI, the overall query time increased significantly (“G* (all graphs, PGI)”), which again points out the importance of sharing computations across graphs.

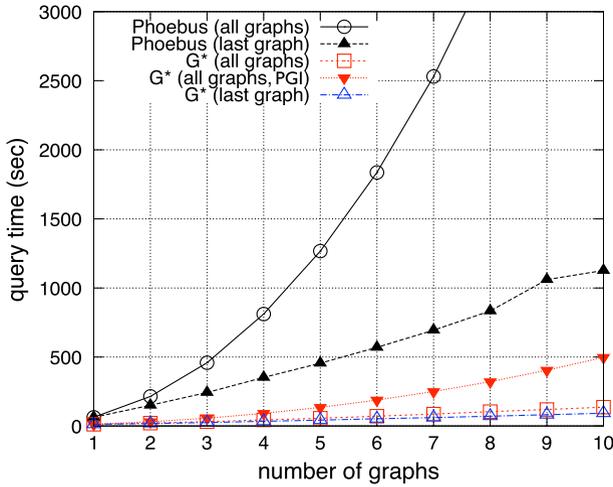


Fig. 21 G* versus Phoebus

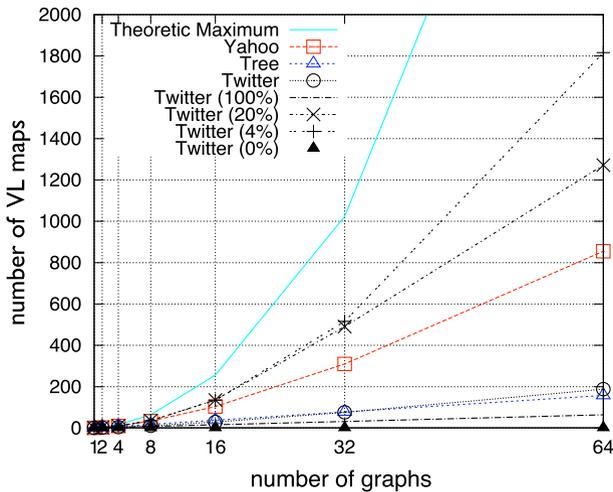


Fig. 22 Number of VL maps

5.2 Impact of indexing

The CGI associates IDs and disk locations of vertices using VL maps (Sect. 3.3). Both lookup overhead and update overhead increase with more VL maps. Figure 22 shows how the number of VL maps varies as the index covers more graphs. Using the Twitter and Yahoo! datasets, we generated two series of 64 cumulative graphs that were hourly snapshots of the underlying network. The number of VL maps increased modestly on these real-world graphs (see “Yahoo!” and “Twitter”) compared to the theoretic maximum (2^N , which is the number of all possible combinations of N graphs).

To carefully examine the impact of similarity between graphs on the number of VL maps (i.e., the robustness of our indexing technique), we also constructed other artificial series of graphs, each of which had a certain fraction (e.g., 0, 4, 20, 100 %) of different edges (introduced by replacing old edges with new edges) compared to its previous graph. In Fig. 22, “Twitter (100%)” shows the result when the graphs had no commonality among them and therefore only one VL map was constructed for each graph. In Fig. 22, the largest number of VL maps were created when 4 % of the edges were changed in each graph.

As explained in Sect. 3.4, we can control the cost of managing VL maps by grouping graphs and constructing a separate index for each group. Figures 23, 24, and 25 show

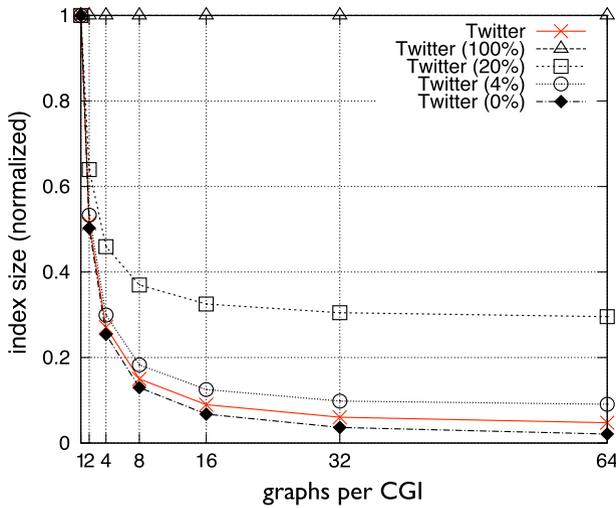


Fig. 23 Index size

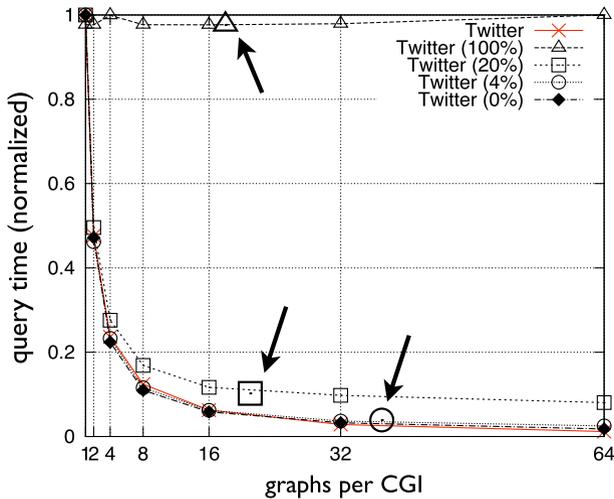


Fig. 24 Query time: average degree

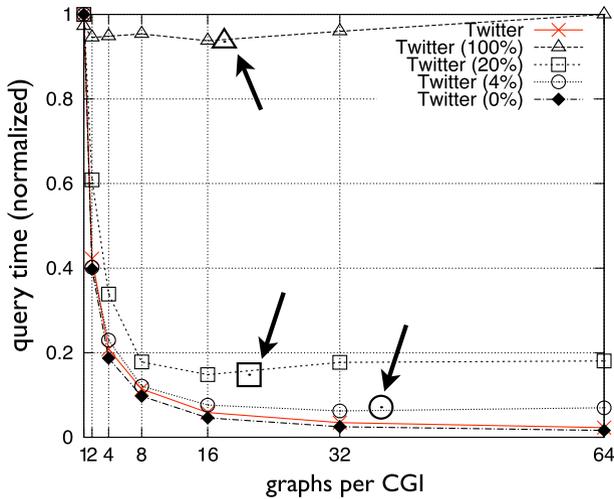


Fig. 25 Query time: clustering coefficient

how the index size and query time vary as the number of graphs managed by each index increases. Figure 25 shows that the query execution time tends to decrease and then increase (compare the query execution times on “Twitter (20 %)” at 16 graphs per CGI and 64 graphs per CGI) as more graphs are indexed together. The reason is that the cost of managing VL maps increases with more graphs.

Our index splitting method (Sect. 3.4) can automatically determine the number of graphs that each CGI needs to cover. In Figs. 24 and 25, the arrows indicate the actual query execution times that our technique achieved on different sets of graphs. This result shows that our index splitting technique makes reliable decisions while achieving near optimal performance. From Figs. 24 and 25 we can also see that, despite its inherent design for shared data storage and computation, G* does not pay any noticeable penalty when there is no commonality between graphs (see “Twitter (100 %)”). In this case, the CGI naturally converges to the PGI, which ignores commonalities among graphs and therefore keeps one VL map for each graph.

5.3 Scalability

This section shows G*’s scalability. These results were obtained by running the first three queries in Fig. 4 on a varying number of G* servers. Among the three datasets in Table 3, only the Tree dataset was used for the shortest distance query. In this case, the root of the tree was selected as the source vertex. When we used other data sets, it was hard to obtain reliable results since the number of visited vertices varied significantly depending on the choice of source vertex.

In one group of experiments (Figs. 26, 27, 28), we increased both the number of servers and the size of the graph at the same rate. When 64 servers were used, the largest graph for each dataset was created using the entirety of the dataset. Figures 26, 27, and 28 show that G* achieved the highest level of scalability when it executed the

Fig. 26 Twitter scale up

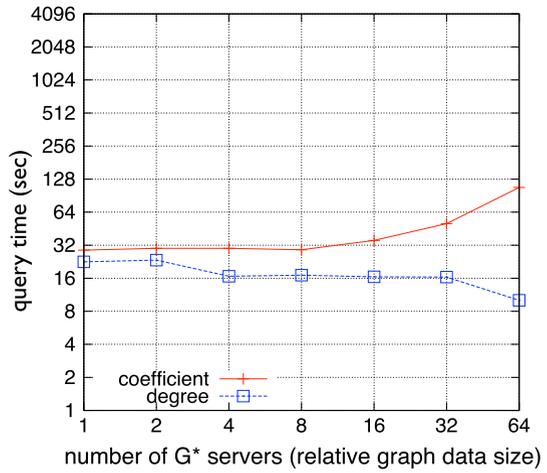


Fig. 27 Yahoo! scale up

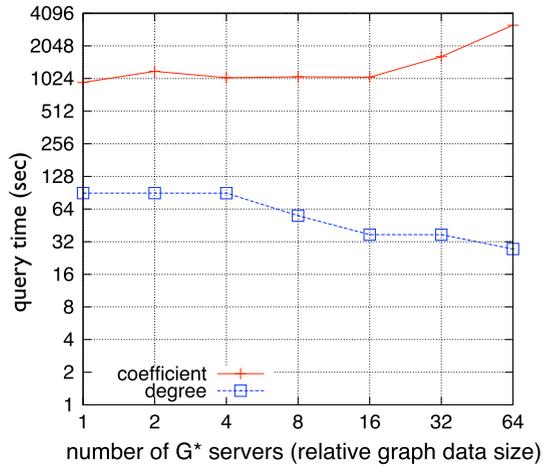
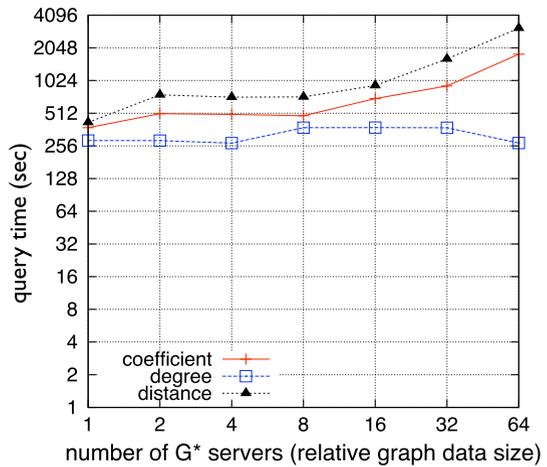


Fig. 28 Tree scale up



vertex degree query. A main reason is that vertex degrees can be computed locally on each server without incurring any network overhead. Further, the number of distinct vertices tends to increase slowly compared to the number of distinct edges. Therefore, each server is assigned a relatively small number of vertices as the number of edges increases in proportion to the number of servers.

When G^* executed the clustering coefficient query, the query time increased gradually as both the number of servers and the amount of data increased. We conjecture that this phenomenon was mainly caused by the increase in the network traffic. We plan to address this issue (i.e., further improve the scalability of G^*) by developing a data distribution technique that can reduce the edges whose end points are on different servers. The result on the shortest distance query is similar to that on the clustering coefficient query.

Figures 29, 30, and 31 show results obtained by creating a graph that contains one million edges and then distributing the graph over an increasing number of servers.

Fig. 29 Twitter speed up

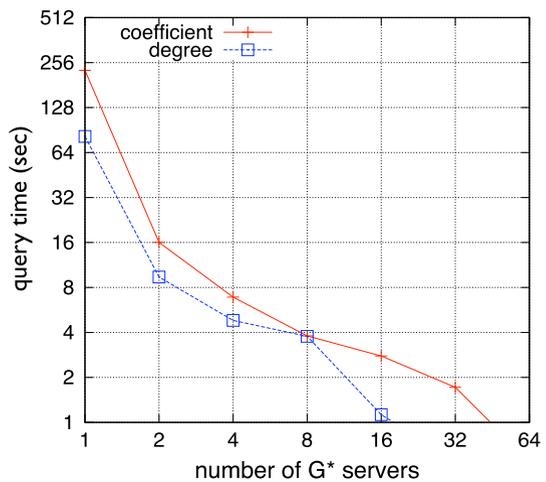


Fig. 30 Yahoo! speed up

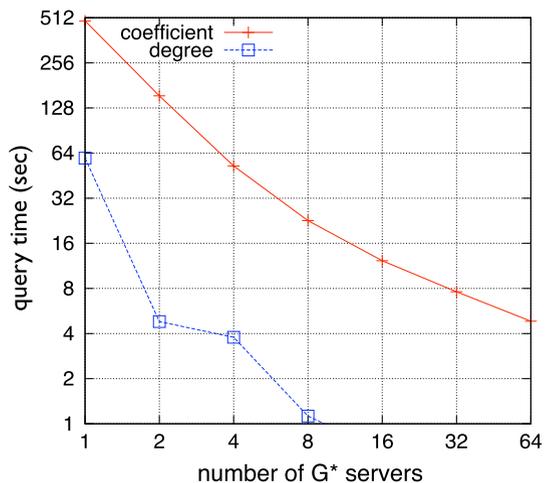
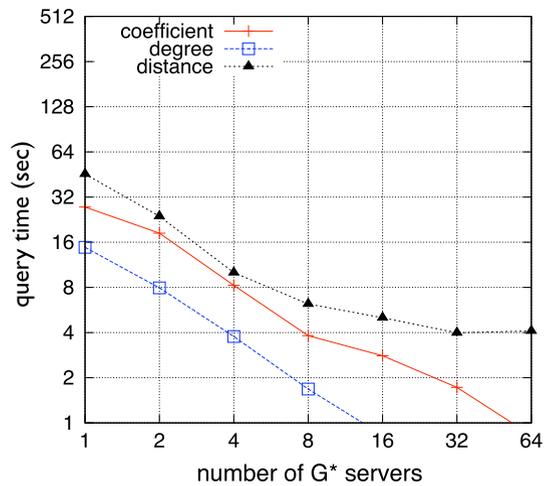


Fig. 31 Tree speed up

In this case, the execution time of each query decreased as more servers were used. The impact of each query on the query execution time is consistent with that in the previous (scale up) experiment.

6 Related work

Google's Pregel is a recent parallel graph processing system [29]. 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 previous graph processing systems such as Parallel BGL [14] and CGM-Graph [10]. Several open-source versions of Pregel are under active development, one of which, Phoebus [36], is compared with G* in Sect. 5. Many open-source systems, like Pegasus [19] are based on Hadoop [1], an open-source implementation of Google's MapReduce [12]. Carnegie Mellon's HADI system [20], 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.

Other recent parallel graph processing systems include Trinity [48], Surfer [11], and Angrapa [47]. In contrast to these systems which process one graph at a time, G* can efficiently execute sophisticated queries on multiple graphs. The performance benefits of G* over the traditional graph processing systems were experimentally demonstrated in Sect. 5.1.

Researchers are working on using multi-core hardware for graph processing. Ligra [40] is a lightweight graph processing framework specifically for shared-memory multi-core machines. Like Trinity, it's a memory-only system and therefore limited by memory size, which G* is not. Acolyte [46] is a similar in-memory graph system (on a smaller parallel scale) from Tsinghua University.

Graph compression techniques typically store *a single graph* by either assigning short encodings to popular vertices [43] 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 [2,7]. These techniques and other previous techniques for compressing graphs [30] and binary relations [5] 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.

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 [15]. Jin et al. have recently presented an efficient indexing technique for reachability queries [18] 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. A demonstration of this work was given at ICDE 2013 [41].

Previous studies on the evolution of dynamic networks were discussed in Sect. 1 of this paper.

7 Conclusion

G^* is a new parallel system for managing a series of large graphs that represent dynamic evolving networks at different points in time. This system achieves scalable, efficient storage of graphs by taking advantage of their commonalities. In G^* , each server is assigned a subset of vertices and their outgoing edges. Each G^* server keeps track of the variation of each vertex and its edges over time through commonality-compressed, deduplicated versioning on disk. Each server also maintains an index for quickly finding the disk location of a vertex and its edges given relevant vertex and graph IDs. Thanks to its space efficiency, this index generally fits in the memory and therefore enables fast lookups.

G^* supports sophisticated queries on graphs using operators that process data in parallel. It provides processing primitives that enable succinct implementation of these operators for solving graph problems. G^* operators process each vertex and its edges once and use the result across all relevant graphs to accelerate queries on multiple graphs. We have experimentally demonstrated the above benefits of G^* over traditional database systems and current graph processing systems.

Acknowledgments This research was supported by NSFCAREER award IIS-1149372 and also supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the "IT Consilience Creative Program" (NIPA-2013-H0203-13-1001) supervised by the NIPA (National IT Industry Promotion Agency).

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. (PVLDB) 2(1), 922–933 (2009)

2. Adler, M., Mitzenmacher, M.: Towards compressing web graphs. In: Proceedings of the 2001 Data Compression Conference (DCC), pp. 203–212 (2001)
3. Alashqur, A.M., Su, S., Lam, H.: OQL: a query language for manipulating object-oriented databases. In: Proceedings of the 15th International Conference on Very Large Data Bases (VLDB), pp. 433–442 (1989)
4. Apache Giraph: <http://incubator.apache.org/giraph/>. Accessed 23 Feb 2014
5. Barbay, J., He, M., Munro, I., Rao, S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 680–689 (2007)
6. Bogdanov, P., Mongiovi, M., Singh, A.K.: Mining heavy subgraphs in time-evolving networks. In: Proceedings of the 11th IEEE International Conference on Data Mining (ICDM), pp. 81–90 (2011)
7. Boldi, P., Vigna, S.: The webgraph framework I: compression techniques. In: Proceedings of the 13th International Conference on World Wide Web (WWW), pp. 595–602 (2004)
8. Bui-Xuan, B.M., Ferreira, A., Jarry, A.: Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. Found. Comput. Sci.* **14**(2), 267–267 (2003)
9. Casteigts, A., Flocchini, P., Quattrociochi, W., Santoro, N.: Time-varying graphs and dynamic networks. In: Proceedings of the 10th International Conference on Ad-hoc, Mobile, and Wireless Networks (ADHOC-NOW), pp. 346–359 (2011)
10. Chan, A., Dehne, F.K.H.A., Taylor, R.: CGMGRAPH/CGMLIB: implementing and testing CGM graph algorithms on PC clusters and shared memory machines. *Int. J. High Perform. Comput. Appl. (IJHPCA)* **19**(1), 81–97 (2005)
11. Chen, R., Weng, X., He, B., Yang, M.: Large graph processing in the cloud. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1123–1126 (2010)
12. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–150 (2004)
13. G* Operator Reference Guide: <http://www.cs.albany.edu/~gstar/operator-reference>. Accessed 23 Feb 2014
14. Gregor, D., Lumsdaine, A.: The parallel BGL: a generic library for distributed graph computations. In: Proceedings of the 4th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC) (2005)
15. Han, W.S., Lee, J., Pham, M.D., Yu, J.X.: iGraph: a framework for comparisons of disk-based graph indexing techniques. *Proc. VLDB Endow. (PVLDB)* **3**(1), 449–459 (2010)
16. He, H., Singh, A.: Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 405–418 (2008)
17. Java Remote Method Invocation (RMI): <http://download.oracle.com/javase/tutorial/rmi/index.html>. Accessed 23 Feb 2014
18. Jin, R., Ruan, N., Dey, S., Yu, J.X.: SCARAB: scaling reachability computation on large graphs. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 169–180 (2012)
19. Kang, U., Tsourakakis, C., Faloutsos, C.: PEGASUS: a peta-scale graph mining system. In: Proceedings of the 9th IEEE International Conference on Data Mining (ICDM), pp. 229–238 (2009)
20. Kang, U., Tsourakakis, C., Appel, A.P., Faloutsos, C., Leskovec, J.: HADI: mining radii of large graphs. *ACM Trans. Knowl. Discov. Data (TKDD)* **5**(2), 8.1–8.24 (2011)
21. Kossinets, G., Watts, D.: Empirical analysis of an evolving social network. *Science* **311**(5757), 88–90 (2006)
22. Kuhlman, C., Kumar, A., Marathe, M., Ravi, S.S., Rosenkrantz, D.: Finding critical nodes for inhibiting diffusion of complex contagions in social networks. In: Proceedings of the European Conference on European Conference on Machine Learning and Principles of Knowledge Discovery in Databases (ECML PKDD), pp. 111–127 (2010)
23. Kumar, R., Novak, J., Tomkins, A.: Structure and evolution of online social networks. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 611–617 (2006)
24. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (USENIX), pp. 31–46 (2012)

25. Lahiri, M., Berger-Wolf, T.Y.: Structure prediction in temporal networks using frequent subgraphs. In: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining (CIDM), pp. 35–42 (2007)
26. Leskovec, J., Backstrom, L., Kumar, R., Tomkins, A.: Microscopic evolution of social networks. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 462–470 (2008)
27. Leskovec, J., Kleinberg, J.M., Faloutsos, C.: Graphs over Time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 177–187 (2005)
28. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: GraphLab: a new framework for parallel machine learning. In: Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI), pp. 340–349 (2010)
29. Malewicz, G., Austern, M., Bik, A., Dehnert, J., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 135–146 (2010)
30. Navlakha, S., Rastogi, R., Shrivastava, N.: Graph summarization with bounded error. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 419–432 (2008)
31. Neely, M.J., Modiano, E., Rohrs, C.E.: Dynamic power allocation and routing for time varying wireless networks. In: Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications IEEE Societies (INFOCOM) (2003)
32. Neo4j: <http://neo4j.org/>. Accessed 23 Feb 2014
33. Nicosia, V., Tang, J., Musolesi, M., Russo, G., Mascolo, C., Latora, V.: Components in time-varying graphs. CoRR abs/1106.2134 (2011)
34. Pan, R.K., Saramäki, J.: Path lengths, correlations, and centrality in temporal networks. CoRR abs/1101.5913 (2011)
35. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf, Raleigh (2008)
36. Phoebus: <https://github.com/xslogic/phoebus>. Accessed 23 Feb 2014
37. PostgreSQL 9.0: <http://www.postgresql.org/>. Accessed 23 Feb 2014
38. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. Proc. VLDB Endow. (PVLDB) 4(11), 726–737 (2011)
39. Santoro, N., Quattrociochi, W., Floccini, P., Casteigts, A., Amblard, F.: Time-varying graphs and social network analysis: temporal indicators and metrics. CoRR abs/1102.0629 (2011)
40. Shun, J., Blelloch, G.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 135–146 (2013)
41. Spillane, S., Birnbaum, J., Bokser, D., Kemp, D., Labouseur, A., Olsen Jr., P., Vijayan, J., Hwang, J.H.: A demonstration of the G* graph database system. In: Proceedings of the 29th International Conference on Data Engineering (ICDE), pp. 1356–1359 (2013)
42. Stanford Large Network Dataset Collection: <http://snap.stanford.edu/data/>. Accessed 23 Feb 2014
43. Suel, T., Yuan, J.: Compressing the graph structure of the web. In: Proceedings of the 2001 Data Compression Conference (DCC), pp. 213–222 (2001)
44. Tan, C., Tang, J., Sun, J., Lin, Q., Wang, F.: Social action tracking via noise tolerant time-varying factor graphs. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 1049–1058 (2010)
45. Tang, J., Musolesi, M., Mascolo, C., Latora, V.: Temporal distance metrics for social network analysis. In: Proceedings of the 2nd ACM Workshop on Online Social Networks (WOSN), pp. 31–36 (2009)
46. Tang, Z., Lin, H., Li, K., Han, W., Chen, W.: Acolyte: an in-memory social network query system. In: Proceedings of the 13th International Conference on Web Information Systems Engineering (WISE), pp. 755–763 (2012)
47. The Angrapa package: http://people.apache.org/~edwardyoon/site/hama_graph_tutorial.html. Accessed 23 Feb 2014
48. Trinity: <http://research.microsoft.com/en-us/projects/trinity/>. Accessed 23 Feb 2014
49. Twitter Streaming API: <https://dev.twitter.com/docs/streaming-apis/streams/public>. Accessed 23 Feb 2014

50. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database: a data provenance perspective. In: Proceedings of the 48th Annual Southeast Regional Conference (SE), pp. 42.1–42.6 (2010)
51. Yahoo! Network Flows Data: <http://webscope.sandbox.yahoo.com/catalog.php?datatype=g>. Accessed 23 Feb 2014
52. Zhao, P., Han, J.: On graph query optimization in large networks. Proc. VLDB Endow. (PVLDB) **3**(1), 340–351 (2010)