Distributed Graph Snapshot Placement and Query Performance in a Data Center Environment

Alan G. Labouseur
School of Computer Science and Mathematics
Marist College
Poughkeepsie, NY 12601
Alan.Labouseur@Marist.edu

Justin Svegliato
School of Computer Science and Mathematics
Marist College
Poughkeepsie, NY 12601
Justin.Svegliato1@Marist.edu

Jeong-Hyon Hwang
Department of Computer Science
University at Albany
State University of New York
Albany, NY 12222
jhh@cs.albany.edu

***Abstract*—Most real-world networks (including financial, health, transportation, social, citation, and sensor networks) evolve over time. Their evolution can be modeled as a series of *graph snapshots* that represent those networks at different points in time. Our distributed dynamic graph database, G*, provided efficient cluster-based storage and querying of graph snapshots by taking advantage of their commonalities. With the modern Internet of Things, however, the environment of continuously generated graph snapshots imbues classic challenges of data distribution with renewed significance. In response to this, we have extended G* to address these new challenges. In this paper we examine two snapshot placement schemes and their effects on query performance in a cloud / data center environment.**

***Index Terms*—Evolving Networks, Graph Analytics, Graph Databases, Distributed and Parallel Systems, Internet of Things**

I. Introduction

We are surrounded by continuously evolving financial, health, transportation, social, citation, and sensor networks [1]. By modeling these networks as graphs whose vertices represent entities and edges represent relationships between entities, network evolution can be captured and harnessed for analytic purposes by recording periodic snapshots of these graphs. Data scientists using these *graph snapshots* can analyze network evolution to discover trends and insights crucial to many fields within the Internet of Things, including networks of electronic health records, geolocation trackers, diagnostic data from connected devices (e.g., smart watches), and more.

While there are several single-graph systems available today (Google's Pregel [2], Microsoft's Trinity [3], Stanford's GPS [4], the open source Neo4j [5], and others), they lack support for efficiently managing the continuously generated graph snapshots inherent in today's Internet of Things. We built G* [6], [7], our distributed system for managing series of graph snapshots, to efficiently store and manage graph snapshots on multiple servers by taking advantage of commonalities among them. We have enhanced G* [8], [9] with today's environment in mind to more efficiently support queries on evolving networks by carefully distributing the data.

Accelerating queries by distributing data over multiple workers has been a popular approach in parallel databases [10] and distributed systems [11]. Techniques for partitioning individual graphs to facilitate parallel computation have also been developed [4], [12], [13], [14]. However, distributing a series of graph snapshots over multiple workers raises new challenges. In particular, it is not desirable to use traditional graph partitioning techniques that consider only one graph at a time and incur high overhead given a large number of vertices and edges. Furthermore, simply distributing each snapshot on all workers may not be appropriate either. This is the problem of **snapshot distribution**: How do we distribute snapshots of an evolving network among our available workers in order to efficiently process various types of queries?

In this paper we examine two approaches to the snapshot distribution problem: *Shared Nothing* placement, where the vertices comprising each snapshot are stored on a single worker, and *Shared Everything* placement, where the vertices comprising each snapshot are shared among all workers. We show that approaching the snapshot distribution problem in this manner results in a system where, for queries involving significant communication among vertices (e.g., PageRank), when multiple evolutionary snapshots are queried together it is more advantageous to store each snapshot on fewer workers. At the same time, we show that for queries involving little communication among vertices (e.g., average degree), when multiple snapshots are queried together, queries on both placements perform consistently.

Key contributions of this work include the following:

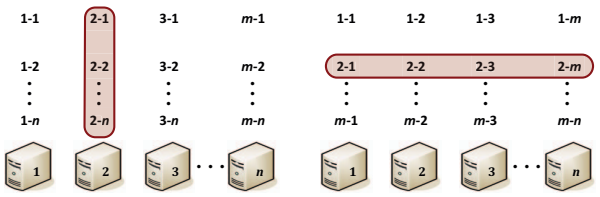
- design and implementation of techniques supporting custom-partitioned graphs in distributed environments
- experimental results showing consistent average degree query execution across placements
- experimental results showing PageRank query execution speedup in *Shared Nothing* placements

II. Background

A. The G* Database Before

G* is a distributed graph database consisting of one *master* server directing many *worker* servers with one or more *client* systems that exchange messages with the *master* to make it go. Individual vertices and their incident edges (collections of which comprise snapshots) are distributed over the workers in a manner controlled by a hash function. For every vertex stored in G*, a hash on its *vertex* ID computes the ID of the worker

978-1-4673-9795-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CSCI.2015.24



(a) Shared Nothing (b) Shared Everything

Fig. 1. **Vertex Placement**. Given m snapshots in the evolution of graph α stored over n workers, Fig. 1(a) shows the "column-oriented" *Shared Nothing* vertex placement where all of the vertices for each of the m snapshots (e.g., graph 2) are stored on a single worker. Fig. 1(b) shows the "row-oriented" *Shared Everything* vertex placement where the vertices comprising each snapshot (e.g., graph 2) are shared among the n workers.

on which it is stored. Each worker efficiently stores its data by taking advantage of commonalities among graph snapshots so that all distinct versions of each vertex are stored on disk only once regardless of how many snapshots they belong to. To track these vertex versions, each worker maintains a *Compact Graph Index* that maps each combination of *graph* ID and *vertex* ID to the disk location storing the corresponding vertex version. To process queries, the master constructs a network of query operators among the workers, who process their data in parallel using a Bulk Synchronous Parallel [2] (BSP) superstep strategy. Further details of our storage and query techniques can be found in our prior publications [6], [7].

B. The Snapshot Distribution Problem

Definition: Given a series of graph snapshots $\{G_i(V_i, E_i) : i = 1, 2, \cdots\}$, n workers, and a set of queries Q on some or all of the snapshots, find a distribution $\{V_{i,w} : i = 1, 2, \cdots \wedge w = 1, 2, \cdots, n\}$ that minimizes $\sum_{q \in Q} time(q, \{V_{i,w}\})$ where $V_{i,w}$ denotes the set of vertices that are from snapshot $G_i(V_i, E_i)$ and that are assigned to worker w, and $time(q, \{V_{i,w}\})$ represents the execution time of query $q \in Q$ on the distributed snapshots $\{V_{i,w}\}$ satisfying (1) $\cup_{w=1}^{n} V_{i,w} = V_i$ (i.e., the parts of a snapshot on all workers cover the original snapshot) and (2) $V_{i,w} \cap V_{i,w'} = \emptyset$ if $w \neq w'$ (i.e., workers are assigned disjoint parts of a snapshot).

In this paper we consider an instance of this problem where the set of queries Q contains two elements: *PageRank* and *average degree* for all vertices of one or all snapshots. We examine the execution time for these queries in two vertex placement configurations: *Shared Nothing* and *Shared Everything*. *Shared Nothing* vertex placement stores all of the vertices comprising each snapshot on **one** of the n workers. See Fig. 1(a). *Shared Everything* vertex placement stores the vertices comprising each snapshot so that they are shared among **all** of the n workers. See Fig. 1(b).

III. The G* Database Now

A. Vertex Placement Modes

Recall from Section II-A that *worker* IDs are computed as a hash of *vertex* ID. This makes determining which worker stores a given vertex fast and easy. There are, however, drawbacks to this approach because the hash determining *worker* ID is based solely on *vertex* ID. One drawback is that the same worker will store all versions of a given vertex for all snapshots, regardless of *graph* ID. This may lead to unbalanced storage, limiting G*'s ability to make efficient use of its workers' CPU, storage, and bandwidth resources. Another drawback is that we have no explicit control over what worker stores a given vertex for a given graph snapshot. This prevents us from addressing the snapshot distribution problem because we **must** be able to explicitly specify on which worker to place a given vertex for a given graph in order to support custom partitioning.

Our solution to the drawbacks mentioned above was to implement two global *Vertex Placement Modes* (VPM) in G*: VPM-Hash and VPM-Custom. The entire system (master, workers, clients) runs in either VPM-Hash or VPM-Custom. VPM-Hash is the default wherein G* works as noted in Section II-A.

While operating in VPM-Custom, G* does not compute *worker* IDs as a hash of *vertex* ID. Rather, G* keeps a (*graph* ID, *vertex* ID)→*worker* ID map in each client. We enhanced the master to accept vertex update messages from clients specifying the *worker* ID on which to store any given (*graph* ID, *vertex* ID) pair. All vertex create and update messages come from a client and go through the master, thus enabling system-wide storage and management in VPM-Custom without impacting our underlying infrastructure.

We have written our own graph partitioner to support the *Shared Nothing* and *Shared Everything* partitions used in this paper. Because of the precise control over vertex placement enabled by VPM-Custom operation, G* can take advantage of other graph partitioners like METIS [12] and new graph partitioning utilities as needs arise.

It is important to note two complications introduced by VPM-Custom. First, there is now overhead when launching a client because it has to load (*graph* ID ,*vertex* ID)→*worker* ID maps from all workers. Second, recall from Section II-A that queries are submitted by a client and processed by the workers in parallel. The workers do not have access to the clients' (*graph* ID, *vertex* ID)→*worker* ID maps. This impacts queries like PageRank because of the need to exchange messages among many vertices (e.g., to calculate the impact of back-links) in order to compute a value for each individual vertex. Queries like average degree are unaffected because they need only look at each vertex by itself (e.g., to count the degree).

B. Exclusion Cache

We modified our message passing mechanism to address the problem of workers being unable to determine what vertices **other** workers store in VPM-Custom. Recall from Section II-A that G* workers execute queries using the BSP model wherein workers exchange messages in supersteps. In VPM-Custom operation, when a worker needs data about a vertex it does not store, it sends requests to all other workers because it can neither compute the appropriate *worker* ID nor look it up (because those maps are stored in the client). This results in a message passing explosion.

We address the message passing explosion by having each worker keep a cache of messages it receives from **other** work-

ers requesting updates for a given (*graph* ID, *vertex* ID) pair. We call this the *exclusion cache*. This cache accumulates a list of workers to avoid asking about various (*graph* ID, *vertex* ID) pairs. For example, if worker 1 receives a message from worker 2 asking for data about (*graph* ID α, *vertex* ID $2-3$) (which is stored on worker 3 in Figure 1(b)) then worker 1 knows that worker 2 does not store (*graph* ID α, *vertex* ID $2-3$) because if it did, it would not have asked. Later, when worker 1 needs data about (*graph* ID α, *vertex* ID $2-3$) it knows not to request it from worker 2 or any of the other workers who had previously asked about it.

In this manner, after a few supersteps, each worker has a set of other workers **not** to ask about many of the (*graph* ID, *vertex* ID) pairs it does not store itself, thus reducing overhead.

One implication of the *exclusion cache* is that the first time a query is run (when the cache is empty) it will have a longer execution time (due to message passing overhead) than in subsequent runs when the cache is populated. After the first execution of a given query, and until the workers are restarted, query performance for that query remains improved. This behavior is shown in Section V.

IV. Experiments

We ran our experiments on IBM PureFlex hardware in our data center combining CPU, storage, networking, and virtualization into a single system optimized for the cloud. Data center environments like this are common among the industries of the Internet of Things today.

Hardware: We used three Intel blade servers in our IBM PureFlex. Each blade was equipped with two Intel 2.9 GHz Xeon E5-2690 CPUs of 8 cores each for 16 cores total, 131GB RAM, and a dedicated NIC to a 20TB storage area network.

Virtual Machines: We installed an Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-57-generic x86 64) virtual machine on each of our three blades. Each was configured with access to all 8 cores, a 70GB partition of the SAN storage, and had the OpenJDK Runtime Environment (IcedTea 2.5.5) installed.

G* Database: We configured G* with one master and 23 workers, each assigned to a single core via the *taskset* command. The VM on the first of our three blades (Gstar-VM-1) ran the master and workers 0 – 6. Gstar-VM-2 was home to workers 7 – 14, with Gstar-VM-3 running workers 15 – 22.

Data Sets: We experimented on 23 evolutionary snapshots of full, complete, binary tree graphs with vertex counts of 1K, 2K, 4K, and 8K in two placement configurations each: *Shared Nothing* and *Shared Everything*.

Queries: We experimented with PageRank and average degree queries. PageRank queries require many vertices to be examined in order to compute the value for any single vertex. Average degree queries need only examine one vertex at a time. These two queries, therefore, represent both ends of the query spectrum. See our prior publication [6] for details of these query operators and how we implemented them in G*.

V. Results and Discussion

We executed PageRank and average degree queries on *Shared Nothing* and *Shared Everything* placements for each of our test data sets. Each query was executed five times and the resulting execution times averaged.

In our first experiment we ran PageRank queries on our 1K *Shared Nothing* data set (23 snapshots of 1023 vertices and 1022 edges each stored in *Shared Nothing* placement over 23 workers (i.e., each snapshot on its own worker)). The execution times, in milliseconds, for those five queries were as follows: 10645, 7979, 6915, 6906, and 6917, with an average time of 7872.4ms. We can see the implications of the cache, mentioned in Section III-B, manifested here. During the first execution of this query the workers did not have exclusion cache data about other workers. Subsequent queries with warm exclusion cache executed considerably faster. This pattern of execution time repeated in PageRank queries for all of our data sets, regardless of size.

For our second experiment we ran PageRank queries on our 1K *Shared Everything* data set (23 snapshots of 1023 vertices and 1022 edges each stored in *Shared Everything* placement over 23 workers (i.e., each snapshot is spread out over 23 workers)). The execution times, in milliseconds, for those five queries were as follows: 14059, 8157, 8053, 8023, and 8024, with an average time of 9263.2ms. Again we see the effect of warm cache in runs 2 – 5 over the cold cache of run 1.

Of particular interest is the *relative speedup* we observe for the PageRank query on all snapshots in *Shared Nothing* placement (7872.4ms average) compared to *Shared Everything* placement (9263.2ms average).

Let us define *relative speedup* as average query execution time in *Shared Everything* placement (τ_{se}) divided by average query execution time in *Shared Nothing* placement (τ_{sn}). I.e., $\tau_{se} \div \tau_{sn} = 1.18$ in this case. (There are no units because this is a relative measure.) Table I summarizes our findings for PageRank queries. Note the speedup we observed for PageRank queries on all snapshots in *Shared Nothing* placement compared to *Shared Everything* placement.

In our third experiment we ran average degree queries on our 1K *Shared Nothing* data set. The execution times, in milliseconds, for those five queries were as follows: 5007, 5008, 4011, 5006, and 5015, with an average of 4809.4ms. Note that we do not observe any cache effects in these query times because computing the average degree for a vertex does not require exchanging messages with other vertices.

For our fourth experiment we ran average degree queries on our 1K *Shared Everything* data set. The execution times, in milliseconds, for those five queries were as follows: 5087, 5090, 4043, 5087, and 5081, with an average of 4877.6ms. Once again we see no cache effects, as vertex placement does not alter the nature of queries (just their execution time).

Of particular interest in these two experiments is the observation that average degree queries perform consistently in three out of four of our scenarios. Average degree queries take approximately the same amount of time in *Shared Everything*

PageRank Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one	7184.8ms	5849.6ms	1.23
1K vertices	all	9263.2ms	7872.4ms	1.18
2K vertices	one	9267.2ms	7619.6ms	1.22
2K vertices	all	11395.8ms	9319.4ms	1.22
4K vertices	one	9305.0ms	7938.4ms	1.17
4K vertices	all	13637.0ms	9408.8ms	1.45
8K vertices	one	9823.8ms	9857.4ms	1.00
8K vertices	all	17795.2ms	11091.0ms	1.60

TABLE I
IMPACT OF SNAPSHOT DISTRIBUTION - PAGERANK QUERIES

Avg. Degree Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one	4640.6ms	2977.2ms	1.56
1K vertices	all	4877.6ms	4809.4ms	1.01
2K vertices	one	5892.4ms	4308.8ms	1.37
2K vertices	all	6343.6ms	6291.2ms	1.01
4K vertices	one	5497.4ms	4617.0ms	1.19
4K vertices	all	6369.8ms	6284.4ms	1.01
8K vertices	one	5787.6ms	4721.4ms	1.23
8K vertices	all	6605.4ms	6636.6ms	1.00

TABLE II
IMPACT OF SNAPSHOT DISTRIBUTION - AVG. DEGREE QUERIES

placement regardless of whether we query one snapshot or all snapshots. This shows the benefits of parallelism when there is little message passing overhead. The single case where we see significant speedup (of 1.56) is when computing the average degree for all vertices of a single snapshot in *Shared Nothing* placement. In this case there is no message passing because the vertices to be examined all reside on the same worker. Table II summarizes our findings for average degree queries.

We observe that results from our 2K, 4K, and 8K data sets follow a similar pattern to those discussed and observed in our 1K data set. The performance of average degree queries on all snapshots is strikingly consistent across placements. We do see some variation, in the form of speedup, in PageRank queries on all snapshots. These experimental results, along with the techniques for supporting custom-partition graphs discussed in Section III, form the contributions of this paper.

VI. CONCLUSIONS

Even in a data center environment with high-end resources combining multiple CPUs, large amounts of RAM, massive storage, and fast networking in powerful systems optimized for the cloud, we still see significant benefits of careful snapshot placement for certain types of queries. Specifically, we have shown that, for queries involving significant communication among vertices (e.g., PageRank), when multiple snapshots are queried together it is more advantageous to store each snapshot on fewer servers, as long as the overall queried data are balanced over all servers to avoid hurting the performance of queries that do not involve significant communication among workers (e.g., average degree).

VII. FUTURE WORK

In our next paper we will be experimenting with larger and different data sets, including "long-tail" Barabasi-Albert graphs and social graphs generated by the Linked Data Benchmark Council [15] benchmark tool. We are also looking at increasing the resolution of our execution time measurements to record data at the superstep level. This could reveal insights into the effectiveness of exclusion cache compared to JVM caching. We are developing techniques for finding hybrid placements that, for certain graphs, might perform better than strictly *Shared Nothing* or strictly *Shared Everything* placements. Finally, further research comparing our data center results with those obtained from experimenting on the 64-core server cluster we used in our prior work may suggest insight into graph query performance on clusters of connected computers with no shared resources as compared to data center environments with many shared resources.

ACKNOWLEDGMENTS

This research was conducted in part on equipment funded by the New York State Regional Economic Development Initiative, CFA 18180, 2012. Earlier work was supported by NSF CAREER Award IIS-1149372. The authors wish to thank the IBM/Marist Joint Study students for their help and support.

REFERENCES

[1] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal, "PageRank on an Evolving Graph," in *KDD*, 2012, pp. 24–32.
[2] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," 2010, pp. 135–146.
[3] Trinity, http://research.microsoft.com/en-us/projects/trinity/.
[4] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," in *SSDBM*, 2013.
[5] Neo4j The Graph Database, http://neo4j.org/.
[6] A. Labouseur, J. Birnbaum, J. Olsen, PaulW., S. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han, "The G* graph database: efficiently managing large distributed dynamic graphs," *Distributed and Parallel Databases*, vol. 33, no. 4, pp. 479–514, 2015.
[7] S. R. Spillane, J. Birnbaum, D. Bokser, D. Kemp, A. Labouseur, P. W. Olsen Jr., J. Vijayan, and J.-H. Hwang, "A Demonstration of the G* Graph Database System," in *ICDE*, 2013, pp. 1356–1359.
[8] A. G. Labouseur, P. W. Olsen, K. Park, and J. Hwang, "A demonstration of query-oriented distribution and replication techniques for dynamic graph data," in *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, 2014, pp. 127–130.
[9] A. G. Labouseur, P. W. Olsen, and J. Hwang, "Scalable and robust management of dynamic graph data," in *Proceedings of the First International Workshop on Big Dynamic Distributed Data, Riva del Garda, Italy, August 30, 2013*, 2013, pp. 43–48.
[10] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, "Gamma - A High Performance Dataflow Database Machine," in *VLDB*, 1986, pp. 228–237.
[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," 2004, pp. 137–150.
[12] G. Karypis and V. Kumar, "Analysis of Multilevel Graph Partitioning," in *SC*, 1995, p. 29.
[13] K. Schloegel, G. Karypis, and V. Kumar, "Graph Partitioning for High Performance Scientific Simulations," Computer Science and Engineering, U. of Minnesota, Technical Report TR 00-018, 2000.
[14] Z. Shang and J. X. Yu, "Catch the Wind: Graph Workload Balancing on Cloud," in *ICDE*, 2013, pp. 553–564.
[15] Linked Data Benchmark Council, http://ldbcouncil.org/benchmarks/snb.