A Hybrid Solution for Mixed Workloads on Dynamic Graphs

Mahashweta Das*
Hewlett Packard Labs
Palo Alto, CA, USA
mahashweta.das@hpe.com

Alkis Simitsis
Hewlett Packard Labs
Palo Alto, CA, USA
alkis.simitsis@hpe.com

Kevin Wilkinson
Hewlett Packard Labs
Palo Alto, CA, USA
kevin.wilkinson@hpe.com

ABSTRACT

The scale and significance of graph structured data today has led to the development of graph management systems that are optimized either for graph navigation requests or graph analytic requests. We present a general purpose graph system that provides high performance concurrently for both navigation and analytic requests. In addition, it supports highly dynamic graphs wherein vertices and edges are added or deleted and properties are modified. Our solution employs a hybrid architecture comprising two graph engines, one for each workload, with a synchronization unit to manage updates and a federation layer to present the hybrid system as a single API to graph applications. We develop a proof-of-concept, describe its implementation in details, and present experimental results that demonstrate its potential.

Keywords

graph management; mixed workload; hybrid architecture

1. INTRODUCTION

Graphs are ubiquitous - from websites to social networks to bioinformatics applications to telcos providing personalized customer services to transportation network to cyber security to workforce management in business organization - we encounter graph structured data every day without realizing it. Naturally, recent times have witnessed the emergence of many new specialized graph management systems for storing, querying, processing, and analyzing graphs. However, these graph engines provide tailored optimizations for different kinds of workloads, algorithms, and executions. Existing graph systems can be broadly classified into two categories [8]: *(i)* navigation or online; and *(ii)* analytic or offline. *Navigation graph management systems* provide high throughput and low latency for short requests that access relatively few graph vertices and edges, e.g., nearest neighbor, reachability query, etc. Typical examples of such systems include graph databases (such as HypergraphDB [2] and Neo4j [3]) and RDF stores (such as Jena and AllegroGraph). *Analytic graph management systems* support long, resource-intensive, analytical computations and iterative batch processing that access a significant fraction of a graph, e.g. PageRank computation, social network analysis, etc. Examples of graph systems of this type are GraphLab [9], Pregel [10], and Giraph [1].

Graphs are a natural way to represent many kinds of enterprise data and analysis of enterprise data is often more easily expressed as a graph computation rather than in SQL. This has led to interesting work [6][14] on graph analytics using relational databases. However, an adaptive enterprise requires both types of workloads, e.g., large numbers of short requests for daily business operations and, to be reactive, longer analytic requests for trend detection, campaign analytics, etc. In addition, there are periodic analytic and reporting needs. Currently, graph engines that perform well on navigation tend not to perform well on analytic, and vice versa. There are several reasons for this:

Graph navigation engines allow updates to the graph and support many concurrent users. Their internal data structures are designed for high throughput requests, accessing and updating a small portion of the graph and these operations conflict with analytic requests that are long-running, consuming most resources on the graph engine. Graph analytic engines store a graph using highly tuned data structures that enable fast traversal of large numbers of vertices and edges. Typically, these data structures are either immutable or have limited support for updates such that the data structures must be completely rebuilt for the best performance. Thus analytic graph engines work best with immutable or slowly changing graphs. Moreover, enterprise data is dynamic. Updates are frequent and bursty, and must not interfere with the analytic requests. No existing graph management system is known to provide high performance concurrently for mixed workloads. A recent graph engine called Trinity [12] support online and offline graph procesing over a distributed memory cloud. However, it is tuned for offline analytics and does not handle updates.

We present a *general purpose graph data management system* called **MAGS** that provide efficient and concurrent processing of graph navigation and graph analytic queries, i.e., mixed workloads for enterprise applications. Each engine stores its own copy of the graph using data structures optimized for a specific workload. Navigation requests and graph updates are directed to one engine, while analytic requests are directed to the second. Periodically, updates are

*Authors are listed alphabetically.

Proceedings of the Fourth International Workshop on Graph Data Management Experience and Systems (GRADES 2016), June 24, 2016, Redwood Shores, USA.

DOI: http://dx.doi.org/XX.XXXX/XXXXXXX.XXXXXXX

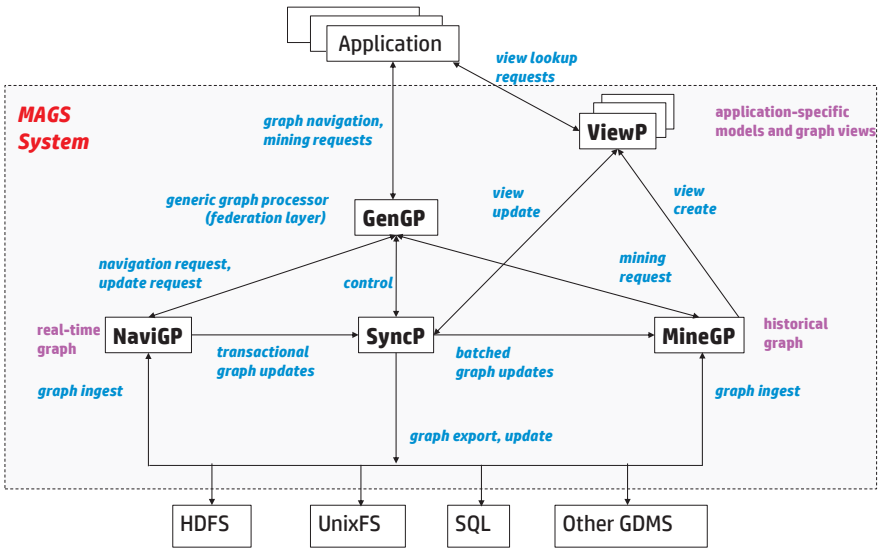

Figure 1: Overview of the hybrid architecture of MAGS System

forwarded from the navigation engine to the analytic engine to keep them in sync. A federation layer presents a unifying API to applications. The key novelty is that by segregating the short navigation requests and updates on the real-time graph from long analytic requests on the historical graph, we can separately tune the two engines to provide the best performance for each workload and prevent updates from interfering with analytic operations. We develop a proof-of-concept that uses the LDBC SNB mixed workload (details later) to demonstrate the potential of the proposed solution.

Sections 2 and 3 describe MAGS architecture and implementation respectively. Section 4 presents an experimental study. Sections 5 and 6 discuss related and future work.

2. THE HYBRID ARCHITECTURE

The hybrid architecture of the our MAGS system is illustrated in Figure 1. It comprises five main modules:

- **GenGP**: A federation module that provides a unifying interface to all graph applications and orchestrates navigation/analytic/update request processing.
- **NaviGP**: A navigation graph engine that processes updates and short navigation requests accessing few vertices and/or edges of the active graph.
- **MineGP**: An analytic graph engine that processes long analytic requests and performs an iterative batch processing over the entire graph until the computation satisfies a stopping criterion.
- **SyncP**: A synchronization module that forward updates from the navigation engine to the analytic graph engine in a transactionally consistent manner.
- **ViewP**: A graph view engine that enables application-specific views of the underlying graph.

MAGS provides applications with different views of the underlying base graph. The first is a real-time view of the graph that reflects transactionally-consistent updates. The second is a historical view that reflects the graph at previous points in time. The third are derived views of the graph that are used to support application-specific purposes.

Next, we discuss the generic details of each module.

Application Interface: The GenGP module is a federation layer that exposes a single API to all graph applications. The data model is a directed, labeled property graph, i.e., vertices and edges may have properties (key-value pairs), edges are directed and vertices and edges may be labeled (have types). This is a general model as it can be used to implement other graph models.

MAGS is intended to accept application requests in a wide variety of interface languages, but currently only SQL is supported (details later). Upon receiving a request, GenGP determines which engine to send it for processing. Short navigation and update queries are sent to NaviGP while analytic requests are sent to MineGP. Currently, a request is processed entirely on one engine. However, that engine may request data remotely from the other engine. GenGP does not itself federate request processing across engines. It employs several techniques in order to orchestrate request processing. The simplest method tags all requests from a particular application or user as one type, or the other (much like database workload managers). In addition, we prototyped a sophisticated classifier that compares features of the input query against a set of rules derived from previously executed queries in order to identify its class, e.g., short navigation, long navigation, analytic, update, etc. The classifier accuracy can be improved by simulating the input query on a small synthetic graph. GenGP is also responsible for system management tasks, e.g., invoking synchronization, dropping old versions of data, view maintenance, etc.

Navigation Requests Processor: The NaviGP module processes short graph requests, i.e., those that access a small fraction of the entire graph. Examples of such queries include nearest neighbor, reachability query, etc. that require very fast response time. It also processes all update requests. In that sense, it is the real-time active graph since it contains the latest version. This engine is tuned for low-latency and high throughput. Popular graph databases like Neo4j [3] and OrientDB [4] are known to support such queries effortlessly.

Analytic Requests Processor: The MineGP module processes all graph requests that are not classified as short or up-

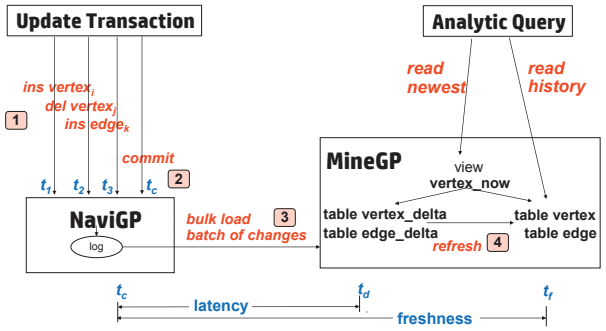

Figure 2: Synchronization unit propagating updates (p_d: load frequency, p_f: refresh frequency)

date. Typically, these are long, possibly iterative and batch requests that access a large fraction of the entire graph. Examples of such queries include Page Rank computation, social network analysis, etc. It may also include requests that cannot be classified as short with high confidence. MineGP operates over the historical graph. If an analytic request requires the most recent graph, MineGP may fetch the latest updates from NaviGP before processing it. Examples of graph engine optimized for analytic requests include GraphLab [9], Pregel [10], Giraph [1], etc.

Synchronization: The SyncP module periodically collects the latest updates in the real-time graph in NaviGP, assembles them into a batch, and bulk loads the changes into MineGP. Log-sniffing is used to collect the NaviGP changes. The bulk load is done transactionally by using versioned tables in MineGP. In this way, synchronization does not interfere with concurrent request processing in NaviGP or MineGP (see Figure 2, which we explain in the next section). The historical graph in MineGP is always behind the real-time graph in NaviGP. However, this delay is tunable and can be relatively short if desired (order of 5-10 seconds), which we believe is sufficient for most applications. This module is also responsible for sending transactionally consistent batched updates to the application-specific derived views of the graph.

View Processor: A common use case in predictive analytics is application-specific models or views where a computation generates a new data structure optimized for fast access later by an application, i.e., the application gets fast access by querying the view rather than the underlying graph engine. An example computation is Page Rank computation. If the viewed data in ViewP is modified in the underlying graph in MineGP, the optimized data structure must either be updated or regenerated. The ViewP module supports this use case. It creates instances of graph-based, application-specific views through an analytic graph request. That view is then informed of changes to the underlying graph as they occur. Alternatively, the view instance may be re-generated by re-executing the original computation.

A possible criticism of our approach is that replicating the entire graph in two engines is wasteful of memory. In fact, complete replication is not necessary. For high performance, only the working set of the graph is needed in the navigation engine. If a NaviGP request needs data outside the current working set, that data can be loaded on-demand from the complete graph in MineGP. So, in a sense, the real-time graph in NaviGP is a cache for MineGP.

Finally, we note that the vast majority of enterprise data resides in relational databases. It is not feasible to replicate all that data in a graph data management system. However, it is important that new graph-based applications have access to this legacy data. MAGS supports this by creating graphical views of relational data using semi-automated mappings, e.g., foreign keys are edges, primary keys identify vertices, etc. When possible, the mappings are bi-directional. In this way, graph applications can seamlessly access and even update relational enterprise data. Hence, our system includes connectors to external systems to import graph data represented in different storage systems or even other graph engines (see the bottom layer of Figure 1).

3. OUR IMPLEMENTATION

In order to study the feasibility of our approach, we built a proof-of-concept using off-the-shelf database systems, along with some custom glue code. In order to choose among candidate systems for NaviGP and MineGP, we conducted a performance analysis of candidate systems using queries from the LDBC Social Network Benchmark (SNB) [5]. The LDBC SNB data generator creates a synthetic graph that mimics a social network. The synthetic data models a Facebook like application with persons, message posts and likes, etc. and includes simple read requests, complex read queries, and batch inserts encoded in SQL. We employ the LDBC SNB interactive workload that has short read, complex read, and updates (in the form of trickle inserts), and add additional bulk load queries and analytic queries (Page Rank computation and single source shortest path computation). Experiments are conducted on a single machine with Intel Xeon E5-2660v2 (40 cores) and 128GB memory using a LDBC SNB graph at scale factor 1, i.e., 3M nodes, 20M edges for 10K persons and 1GB size.

Figure 3 compares the performance of a native graph database, relational database MySQL, and analytic database management system Vertica. Vertica is known to be optimized for processing long iterative analytic queries [6]. Figure 3 shows that it is a good fit for analytic workload, and hence is our MineGP engine. From Figure 3, we also see that MySQL performance for short read queries and updates was adequate and on par with that of graph database. Relational databases as NaviGP and MineGP would enable querying using a common language, i.e., SQL. In addition, we have existing code to read the MySQL log that we can leverage for SyncP. Hence, MySQL is our NaviGP engine. GenGP exposes a RESTful web API to all graph applications and accepts queries in SQL. Figure 3 also validates our hybrid approach, i.e., there does not exist one single best engine optimized for different kinds of workloads.

One important aspect of our implementation is the design of SyncP that propagates updates from NaviGP (i.e., MySQL) to MineGP (i.e., Vertica). We use a table versioning technique as illustrated in Figure 2. Suppose, the real-time active graph in NaviGP receives short update transactions that do one or more of changing an existing vertex, edge or property in the graph. Example queries in the context of LDBC SNB workload are insert new vertex (i.e., new person who just joined the social network), delete existing edge (i.e., person removed a friend in the social network), etc. Figure 2 shows a transaction making three graph changes that commits at time t_c. At this point, the changes are visible in the real-time graph. Periodically,

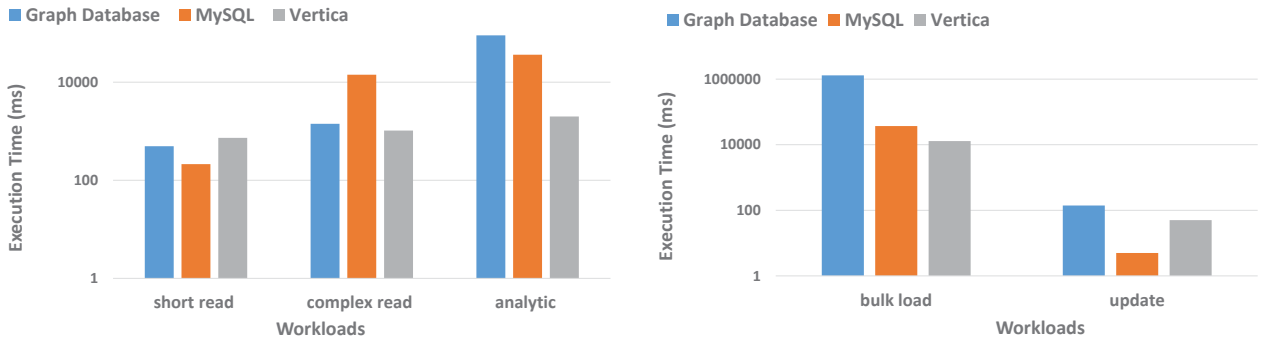

Figure 3: Performances of graph database, in-memory relational database MySQL, and analytic database Vertica for different types of workload

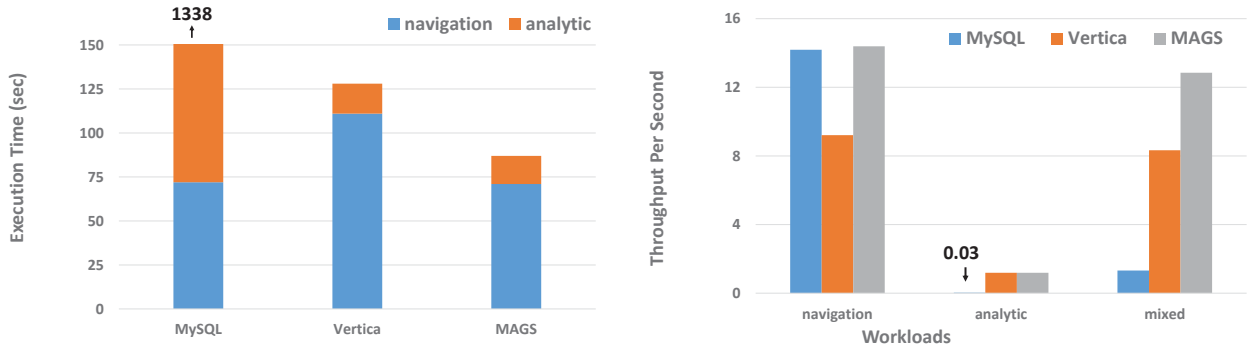

Figure 4: (left): Latency of individual best-of-class engines (MySQL, Vertica) and hybrid engine (MAGS) for mixed workloads; (right): Throughput of individual best-of-class engine (MySQL, Vertica) and hybrid engine (MAGS) for different types of workload

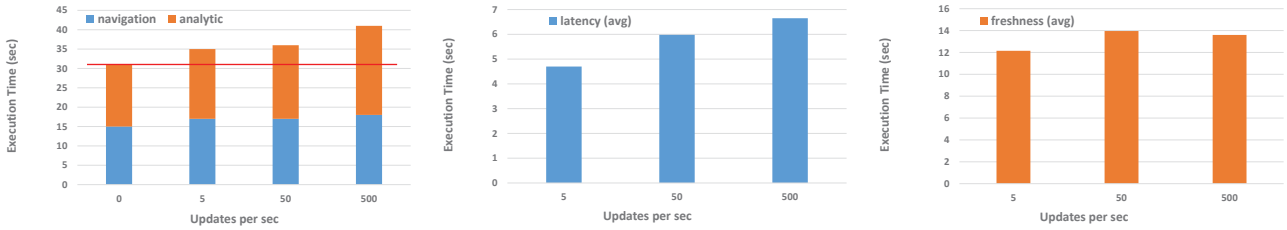

Figure 5: (left): Contention in mixed workloads with varying number of updates for MAGS; (center): Average latency on mixed workloads with a varying number of updates for MAGS; (right): Average freshness on mixed workloads with a varying number of updates for MAGS

SyncP wakes up and batches all graph changes since its last invocation into a single batch load command for MineGP. This is known as the load frequency. These changes are appended to delta tables in MineGP (which completes at time t_d in Figure 2). We define the latency of a change as the time difference between its commit and the load time, i.e., t_d - t_c. Periodically, the delta tables are merged with the full historical tables (which completes at time t_f in Figure 2). This is known as the refresh frequency. We define the freshness of a change as the time difference between its commit and the refresh completion, i.e., t_f - t_c. Both load and refresh frequencies may be varied. Note that applications that require fresher data may read the delta tables (through views provided by MAGS) or may request invocation of a new load/refresh cycle. Experimental results in Section 4 validates how this synchronization model has low impact on workload.

SyncP is responsible for sending transactionally consistent batched updates to the ViewP module too, while MineGP updates the application-specific derived views of the graph at regular intervals. Existing graph query processing systems such as GraphLab is a potential candidate for ViewP. To connect a relational database (Vertica) and a graph engines (GraphLab), we need to either replicate all the relational data in the graph engine or transfer data as needed per request. For a number of reasons related to stability, robustness, performance, scalability, and support for legacy applications, the former does not seem practical, i.e., a graph engine should not replicate all data and functionality of an enterprise. We need to have both engines and a connection between them. But this is not trivial either since it requires moving the data from the relational database to the graph engine, computing the result, and moving it back to the relational database. As a proof-of-concept, we implemented

a fast, in-memory, bi-directional connector between Vertica and GraphLab that uses a shared memory to greatly reduce the data shipping and function shipping overhead between the two engines [7].

4. MIXED WORKLOAD EVALUATION

We have introduced our experimental set up in Section 3. Using the same hardware configuration, same LDBC SNB graph data, and LDBC SNB interactive workload complemented with additional analytic queries, we demonstrate MAGS performance for mixed workload in Figure 4. Our mixed workload comprises 1041 queries of which 1022 are short requests, e.g., short read in LDBC SNB interactive workload and 23 are long requests, e.g, complex reads in LDBC SNB interactive, Page Rank, etc. Since the workload in a social network is usually read-dominated, the mixed workload has a dominance of short query requests. Figure 4 (left) reports the execution time, i.e., latency while Figure 4 (right) reports the number of queries executed per second, i.e., throughput. We observe that MAGS performs better than each individual best-of-class engines, i.e. MySQL (the navigation engine) and Vertica (the analytic engine). Note that, MySQL has very high latency and very low throughput for analytic queries while Vertica has high latency and low throughput for navigation queries. Since MAGS is a hybrid system and encompasses the *best of both worlds*, MAGS has the lowest latency for mixed workload (Figure 4 left) and highest throughput for mixed workload (Figure 4 right).

Next, we measure contention, latency, and freshness of the synchronization procedure illustrated in Section 3 and Figure 3. Recall that MAGS objective is to run mixed workload efficiently and concurrently. While mixed workload comprises of navigation and analytic queries, it may also include updates in the form of inserts and deletes. We refer to it as the update workload (so that mixed workload continues to consist of navigation and analytic queries). The LDBC SNB interactive workload has been extended to include these additional queries, the details of which are described in the Appendix. In addition to the mixed workload of 1041 queries, we employ a update workload of insert/delete queries where each transaction does 15 insert or 15 delete. Keeping in mind the reality of social network, the insert/delete queries comprise message *posts* and associated *likes* and *tags*.

Figure 5 (left) demonstrates that the synchronization procedure has low impact on the execution of the mixed workload, i.e., there is no contention. We compare the time taken by MAGS for mixed workload when there is no update workload (i.e., 0 updates per second) with the time taken when there is an update workload in the background. MAGS takes only 10 additional seconds to execute the mixed workload when there are 500 updates/second in the background. In order to evaluate the latency and freshness of the synchronization procedure, we consider that SyncP periodically propagates updates every 10 seconds and refreshes every 20 seconds. Figure 5 (center) demonstrate average latency on mixed workloads with a varying number of updates where latency is defined as the time between commit to NaviGP and commit the batch of delta changes to MineGP. and Figure 5 (right) demonstrate average freshness on mixed workloads with a varying number of updates where freshness is defined as the time between commit to NaviGP and refresh historical data in MineGP.

5. RELATED WORK

State of the art graph management systems are optimized for different kinds of workloads. Graph analytic engines like GraphLab [9], Pregel [10], Giraph [1], etc. support analytic computations that batch process a large fraction of the entire graph while graph databases like Neo4j [3], HypergraphDB [2], etc. support navigation computations that access fewer nodes and/or edges in the graph. External memory graph processing systems like FlashGraph [15] and in-memory graph processing systems like GEMS [11] focus only on graph algorithms for large graphs. Teradata's graph engine Aster [13] enables native processing of large-scale graph analytic queries, but does not support navigation short query requests. Recently, a distributed in-memory graph system Trinity [12] has been developed that supports low-latency online query processing and high-throughput offline analytics on large graphs. However, Trinity is tuned for offline analytics, does not handle updates, and does not support the workloads concurrently as MAGS does.

6. CONCLUSION AND FUTURE WORK

A significant portion of big data today is graph structured data that captures our daily activities, intents, and interactions. Currently, enterprises use two or more systems to manage their real-time graph, their historical graph, and their derived graphs (views, i.e., application-specific models). MAGS provides a unified framework for managing the different graphs and processing the different graph queries concurrently and efficiently. We present a flexible hybrid architecture that utilizes existing graph navigation engines and graph analytic engines for executing mixed workload efficiently and concurrently. Our intention in this work is not to focus on how specific graph engines perform. Rather we aim at demonstrating that a hybrid solution has its merit.

There are several directions going forward. A next step is to work on a scale-out architecture for the hybrid graph data management system and exploit next-generation hardware for improved latency and throughput. Instead of employing state of the art third party graph engines, designing graph engines optimized for analytic and/or navigation workload and tuned for next generation computing resources may be useful. The system can be extended to handle multiple workloads coming from multiple graph applications. The federation layer that exposes a generic graph API to applications can benefit from a generic graph query language independent of the underlying navigation and analytic engine specifications. Finally, a bi-directional connector between the analytic graph engine and the view processor engine can greatly reduce the data shipping and function shipping overhead between the two engines.

7. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.

[2] HypergraphDB. http://www.hypergraphdb.org/.

[3] Neo4j. www.neo4j.org.

[4] OrientDB. http://orientdb.com/orientdb/.

[5] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *ACM SIGMOD*, 2015.

[6] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph Analytics using Vertica Relational Database. In

IEEE Big Data, 2015.

[7] A. Kalinin, A. Simitsis, K. Wilkinson, and M. Das. VGL: Enabling Graph Computation over Enterprise Data. Technical report, Hewlett Packard Labs, 2016.

[8] A. Khan and S. Elnikety. Systems for Big-Graphs. *PVLDB*, 2014.

[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A New Framework for Parallel Machine Learning. In *UAI*, 2010.

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

[11] A. Morari, V. G. Castellana, O. Villa, J. Weaver, G. T. Williams, D. J. Haglin, A. Tumeo, and J. Feo. GEMS: Graph Database Engine for Multithreaded Systems. In *Big Data - Algorithms, Analytics, and Applications.*, 2015.

[12] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *ACM SIGMOD*, 2013.

[13] D. E. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and Y. Xiao. Large-scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. *PVLDB*, 2014.

[14] A. Welc, R. Raman, Z. Wu, S. Hong, H. Chafi, and J. Banerjee. Graph Analysis: Do we have to Reinvent the Wheel? In *GRADES co-loated with ACM SIGMOD/PODS*, 2013.

[15] D. Zheng, D. Mhembere, R. C. Burns, J. T. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *FAST*, 2015.

APPENDIX

The LDBC Social Network Benchmark (SNB) [5] comprises simple read-only queries, complex read-only queries, and transactional update requests. The update requests (pre-generated by SNB data generator) add a user account, add friendship, add a forum to the social network, create forum membership for a user, add a post/comment, and add a like to a post/comment. They do not actually delete or update content. A graph update is typically defined as changes made to an existing vertex, edge or property in the graph. We extend the LDBC SNB workload to incorporate true graph updates. We prototyped this change by running our true updates as a background process while the standard LDBC workloads are executing, and we compare the impact on performance to the baseline where updates are not performed (see Section 4).

An example of a true graph update is a real world social network scenario where people periodically clean-up their online presence by deleting old posts along with their associated likes and tags. The code snippet for generating the above update is shown below in the OldPost method. Note, to keep the database size constant during our experiments, it was necessary to run a corresponding NewPost transaction at the same frequency as OldPost. This method is not shown. It simply inserts a new post along with associated likes and tags. In the update workload that we consider in Section 4, we have NewPost and OldPost requests where each request does, on average, 15 inserts and 15 deletes, respectively.

```
public class OldPost extends LDBCProcedure {

      public final SQLStmt  stmtDeletePostSQL = new SQLStmt(
            "DELETE FROM " + LDBCConstants.TABLENAME_POST +
            " where ps_postid = ?");

      public final SQLStmt  stmtDeleteLikesSQL = new SQLStmt(
            "DELETE FROM " + LDBCConstants.TABLENAME_LIKES +
            " WHERE l_postid = ?");

      public final SQLStmt  stmtDeletePostTagSQL = new SQLStmt(
            "DELETE FROM " + LDBCConstants.TABLENAME_POSTTAG +
            " WHERE pst_postid = ?");

      // OldPost Transactions
      private PreparedStatement stmtDeletePost = null;
      private PreparedStatement stmtDeleteLikes = null;
      private PreparedStatement stmtDeletePostTag = null;

    public ResultSet run(Connection conn, LDBCWorker w) throws SQLException {

          long postIdMin = xxx; // delete posts starting here
          // w.oldPostCount is the number of successful post deletes in this run
          long postId = postIdMin + w.oldPostCount;

          try
          {
          stmtDeletePost    = this.getPreparedStatement(conn,
                    stmtDeletePostSQL);
          stmtDeleteLikes   = this.getPreparedStatement(conn,
                    stmtDeleteLikesSQL);
          stmtDeletePostTag = this.getPreparedStatement(conn,
                     stmtDeletePostTagSQL);

          stmtDeleteLikes.setLong(1,postId);
          int result = stmtDeleteLikes.executeUpdate();

          stmtDeletePostTag.setLong(1,postId);
          result = stmtDeletePostTag.executeUpdate();

          stmtDeletePost.setLong(1,postId);
          result = stmtDeletePost.executeUpdate();

          } catch(UserAbortException userEx)
          {
              LOG.debug("Caught an expected error in New Post");
              throw userEx;
          }
         finally {
          }
          return null;
    }

}
```

It is important to track latency of propagating updates from the navigation engine to analytic engine. To measure this, we use an auxiliary table, Fresh, that is replicated on both engines. After every 50 requests (any combination of NewPost or OldPost), we execute a method to insert a new row in the Fresh table on NaviGP. The code snippet is shown below in the Fresh method. We then measure the delay until that row appears in MineGP (as propagated by SyncP).

```
public class Fresh extends LDBCProcedure {

      public final SQLStmt  stmtInsertFreshSQL = new SQLStmt(
            "INSERT INTO "+ LDBCConstants.TABLENAME_FRESH +
            " (id) VALUES ( 0 )");

      // Fresh Transaction
      private PreparedStatement stmtInsertFresh = null;

    public ResultSet run(Connection conn, LDBCWorker w) throws SQLException {

          stmtInsertFresh = this.getPreparedStatement(conn, stmtInsertFreshSQL);

          try
          {
               int result = stmtInsertFresh.executeUpdate();
          } catch(UserAbortException userEx)
          {
              LOG.debug("Caught an expected error in Fresh");
              throw userEx;
          }
          finally {
          }
          return null;
     }
}
```