Janus: Transactional Processing of Navigational and Analytical Graph Queries on Many-core Servers

Hideaki Kimura
Hewlett Packard Labs
Palo Alto, CA, USA
hideaki.kimura@hpe.com

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

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

ABSTRACT

Existing scale-up graph engines are tuned for either short, navigational requests (e.g., Nearest-Neighbor) or longer, analytics requests (e.g., PageRank). However, they do not have good performance for both workloads running concurrently. We present Janus, a scale-up graph engine architected for modern, many-core servers with large memory. Janus has excellent scale-up performance on navigational requests, on analytics requests, and on a mixed workload running concurrently both navigational and analytics requests.

1. INTRODUCTION

Graphs applications can be found in many domains like social networks, bioinformatics, telcos, transportation networks, workforce and IT asset management, cyber security, national security, and various scientific computing domains, to name a few. So it is without a surprise that many new specialized graph engines have emerged for storing, querying, processing, and analyzing graphs. And these graph engines provide tailored optimizations for different kinds of workloads, algorithms, and executions.

Existing graph engines may be either scale-up or scale-out. Scale-up engines have excellent performance but the graph is limited by the resources on a single node. Scale-out engines can distribute a large graph across a cluster of nodes but the communication overhead limits scalability [4].

For graph applications, the workloads may be broadly characterized as navigational or analytics. A navigational workload comprises requests that access relatively few graph vertices and/or edges (e.g., Nearest-Neighbor). An analytics workload comprises requests that are resource-intensive and access a large fraction of a graph (e.g., PageRank). In many applications, it is desirable to run both workloads concurrently. In this case, the large, longer requests interfere with the shorter requests and vice-versa.

In general, graph engines that perform well on navigation tend not to perform well on analytics, and vice versa. There are several reasons for this. Navigation oriented graph 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. Analytics requests are long-running, tend to consume most resources on the graph engine, and so degrade service for concurrent navigational requests. On the other hand, analytics oriented graph engines store a graph using highly tuned, read-optimized data structures that enable fast traversal of large numbers of vertices and edges. Graph updates typically require these data structures to be rebuilt. So analytics graph engines work best with immutable or slowly changing graphs. In practice, graph engines are optimized for either analytics or navigation; they are not doing both well and especially, when both workloads run concurrently.

Hence, existing graph engines may be categorized into four quadrants: scale-up vs. scale-out and navigational vs. analytics. There is no single, best engine for all environments.

Emerging computer architectures provide us with an opportunity to build graph engines in a different way. Next-generation server architectures are designed to be equipped with thousands of cores, petabytes of volatile and non-volatile memory, connected via fast interconnect [7]. An application on such a system could maintain a large graph and execute resource-intensive requests concurrently with shorter requests. Consequently, we believe that such an emerging system would be an excellent platform for big graph applications running mixed workloads concurrently and efficiently. This could enable a single, best graph engine. Our work is an attempt toward this goal.

2. JANUS

Janus [1] Graph Engine is a scale-up graph engine designed for many-core, large-memory servers. Janus serves mixed graph workloads, navigational and analytics, as well as insertions and deletions. The key design points of Janus are as follows.

- Janus is designed to concurrently process navigational and analytics requests, and at the same time allow efficient data ingestion.

- Janus is designed to fully utilize the power of modern servers that employ thousands of CPU cores, huge volatile and non-volatile memory, which most likely has non-uniform access cost.

[1]The name is inspired by the Roman god with two faces.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17)
January 8-11, 2017 , Chaminade, California, USA.

2.1 Scalable Transaction Processing for Graphs

Our key idea to satisfy the above goals is leveraging *transaction processing*. Graph analysis usually consists of read-only accesses. Thus, most of existing analytics graph engines have not incorporated transaction processing. Nevertheless, transaction processing can serve key roles in graph analytics for two reasons.

First, transactions allow graph analytics, and also navigational requests, to be consistent in the presence of concurrent data ingestion (e.g., inserts or deletes on edges and vertices). This overcomes the suboptimal approach that some existing graph engines often take, which requires either pausing queries during data ingestion and/or re-populating the entire graph even for small updates.

Second, a graph query executed on a many-core server consists of many transactions. Parallelizing a complex computation onto a large number of CPU cores fundamentally involves concurrency control. Each CPU core has to communicate with other cores to exchange intermediate progress in a consistent yet concurrent fashion. As we shall see in the later sections, these cores often access an overlapping region of data, necessiating serializable coordination for correct results and/or fast convergence. If the internal transactions are not processed in a scalable, transactional engine, the scalability of the entire graph engine will be severely limited. In fact, in our experiments presented in Section 3, we observed exactly this limitation on GraphLab [11] presumably due to its suboptimal transactional processing engine.

To generalize, somewhat surprisingly, the scalability of transaction processing can be the crux of apparently read-only analysis. In order to be parallelized on a modern server environment, such an analysis requires billions of highly conflicting transactions to be processed. This paper focuses on graph processing, but the same challenge would also appear in parallelizing various computationally heavy analysis, such as network intrusion detection and image recognition.

2.2 Architecture Overview

Janus uses a highly scalable data management engine that provides concurrent accesses for both navigational requests and analytics requests. Navigational requests benefit from a locality-aware data management layer that colocates related data (e.g., closely connected vertices) and requests to minimize remote memory traffic. Analytics requests split their frequently extremely heavy work across hundreds of CPU cores that must communicate with each other to consistently share and merge the results of individual computation. In our implementaion, we built Janus on top of FOEDUS [10], a transactional database built for next-generation computer architectures, like HPE's The Machine [7]. FOEDUS is designed for scalable transaction processing on emerging server hardware.

The key principle in Janus is that all data accesses are handled as database transactions, as illustrated in Figure 1.

FOEDUS employs group-commit, which fits well for parallelizing analytics graph requests. Group-commit enables higher throughput at the cost of higher latency for durably committing an individual transaction. However, the higher latency is not a concern in this case because a large number of internal transactions naturally form a group for durability commit.

The optimistic concurrency control (OCC) scheme in FOEDUS also fits well for processing navigational graph requests.

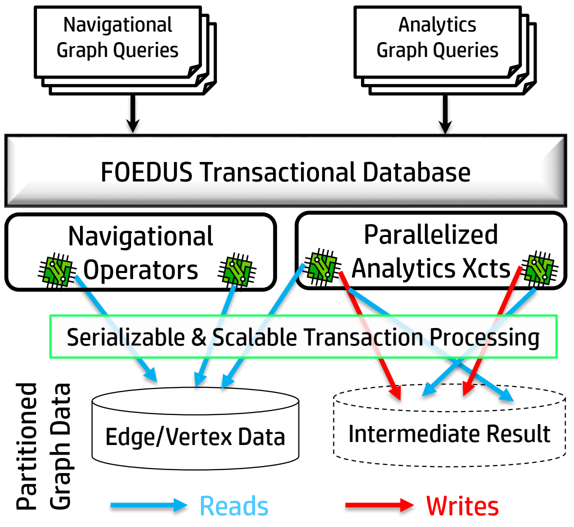

Navigational Graph Queries
Analytics Graph Queries
FOEDUS Transactional Database
Navigational Operators
Parallelized Analytics Xcts
Serializable & Scalable Transaction Processing
Partitioned Graph Data
Edge/Vertex Data
Intermediate Result
Reads
Writes

Figure 1: Janus Architecture Overview

FOEDUS uses a variant of OCC based on *dual pages* [10]. Dual pages allow read-only, serializable transactions (e.g., navigational queries) to immediately commit without unscalable read-locks, durability commit latency, or any chance of aborts.

Next, we provide a detailed view of Janus. We first describe how we store graph data in Janus and then, how we process navigational, analytics, and mixed workloads.

2.3 Storing Data

Janus supports directed, property graphs. It stores a graph as a table VG, named VertexIndexTable (see also Figure 2), with a schema:

$[vid, <meta>, edg\_cnt, <eid_1, meta_1>, ..., <eid_N, meta_N>]$

vid is a vertex id, $<meta>$ stores the metadata of vid (i.e., properties), edg_cnt is the number of outgoing edges (or out-degree) of vid, the $<eid_i, meta_i>$ pair is the metadata of an outgoing edge, and eid_i is a vertex id. Depending on the graph size and the complexity of its topology, the metadata can be stored inline in VG or $meta_i$ can point to the metadata stored elsewhere. Internally, VG is loosely horizontally partitioned.

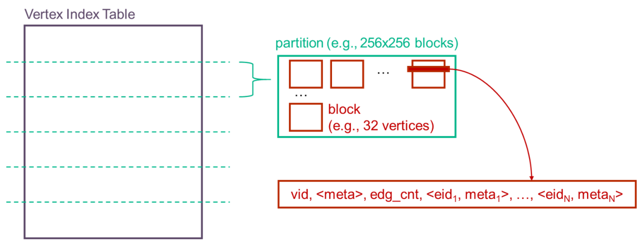
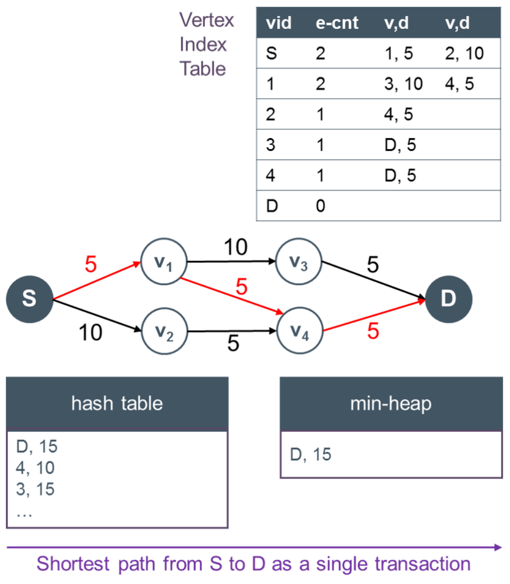

Vertex Index Table
partition (e.g., 256x256 blocks)
block (e.g., 32 vertices)
vid, <meta>, edg_cnt, <eid1, meta1>, ..., <eidN, metaN>

Figure 2: Storage scheme

An example partitioning scheme is depicted in Figure 2. Each partition contains blocks and each block stores 32 vertices (i.e., a partition stores 2M vertices). A transaction may update the metadata (of a vertex or edge) in a row of VG. When this happens, it exclusively locks the row for the duration of the transaction.



vid	e-cnt	v,d	v,d
S	2	1, 5	2, 10
1	2	3, 10	4, 5
2	1	4, 5	
3	1	D, 5	
4	1	D, 5	
D	0		

Figure 3: Example navigational query

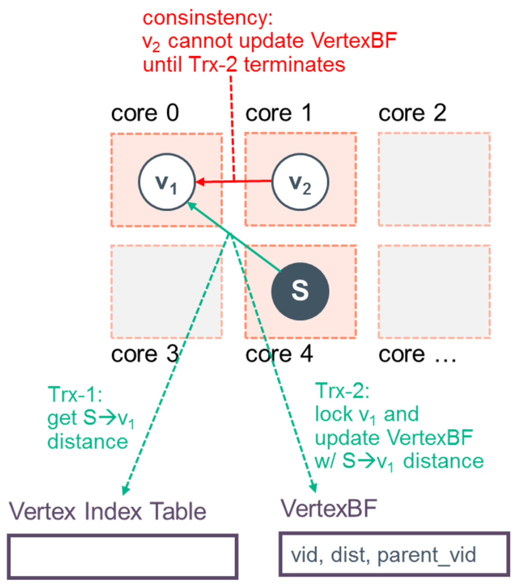

consinstency:
v_2 cannot update VertexBF until Trx-2 terminates
core 0
core 1
core 2
v_1
v_2
S
core 3
core 4
core …
Trx-1: get S→v_1 distance
Trx-2: lock v_1 and update VertexBF w/ S→v_1 distance
Vertex Index Table
VertexBF
vid, dist, parent_vid

Figure 4: Example analytics query

2.4 Navigational Workload

A typical navigational request accesses a small portion of a graph (e.g., only a few vertices and/or edges, and no more than a few dozen vertices and/or edges). For navigational workloads, Janus treats each access to a vertex or to an edge as a single transaction.

For example, assume that we want to compute a typical navigational query that finds a shortest path between a pair of vertices within tens of hops. We employ a standard Dijkstra shortest path algorithm. To better illustrate our approach, see the example shown in Figure 3. Assume a source S and a destination D vertex. We use a hash table and a min-heap for intermediate storage. Starting from S, we use VG (the VertexIndexTable described in the previous section) to get the outgoing edges for S and their respective distances. We update min-heap and store the shortest distances in the hash table. We continue until we find D at the top of the min-heap. Each shortest path query executes as a single transaction in a single thread.

Doing such computations as a single transaction is a unique feature that enables our engine to concurrently run a large number of navigational queries with up-to-date information. And under the hood, FOEDUS transactionally guarantees that the properties and existence of the vertices/edges are not changed by concurrent updates. (Note, that in order to compute the query result, the transaction may update intermediate data in min-heap). A concurrent update on unrelated piece of the graph does not cause any slowdown. A concurrent update on a related piece is detected by Janus, which then immediately retries the query to guarantee serializable result.

2.5 Analytics Workload

A typical analytics query touches virtually all vertices in a graph, often hundreds of millions. This imposes additional challenges to a graph engine. Janus performs such graph computations as a large number of small transactions.

For example, assume a typical analytics computation like single source shortest path (SSSP), which starting from a given vertex (source) finds the shortest path to every other vertex in the graph that is eventually connected with the source. In this case, we employ a standard Bellman-Ford algorithm that can be nicely distributed to compute all shortest paths from a single source vertex. A typical approach is to run many workers that individually calculate and propagate ('relax') distances from the source vertex. After propagating the updated distance information, the worker also communicates with another worker to request further calculation and propagation based on the information the worker updated ('activate'). The query ends when there is no more distance information any worker can relax.

Janus follows the same idea, but with the twist that it runs the calculation, propagation ('relaxation'), and activation phases on a large number of cores as concurrent transactions. The transactions guarantee consistency even when two workers are concurrently updating the same vertex. In particular, we maintain a table, VertexBF, with schema $[vid, distance, parent\_vertex\_vid]$. See also the illustration in Figure 4. Starting from a source vertex S, we use VG to get its neighbors and their distances. At each iteration, we update VertexBF. We build this algorithm in a transactional aware fashion. We partition the vertices space into blocks and assign each such block to a different worker, which is handled by a different core (a thread is attached to a core for NUMA awareness). When S reaches out to a vertex v_1 we use a single transaction to get the distance from VG. Then, we use a different transaction to update that distance in VertexBF, while we lock v_1.

This scheme guarantees transactional consistency (e.g., v_2 cannot update the distance in the meantime) and this fine-grained locking granularity is adding up to get a great speedup comparing to state-of-the-art graph databases that lock larger objects (partitions, tables). On the other hand, this scheme uses billions of transactions even for a single query, like SSSP. But having FOEDUS at the back-end, which is a fast transactional engine, this becomes an advantage for graph computations.

Table 1: Data sets

data size	vertices	edges
small	2.1M	33.6M
medium	96.5M	1.61B
large	402.7M	6.46B

2.6 Mixed Workloads

Some graph systems segregate navigational and analytics requests and send them to different engines. They also depend on different partition schemes for navigational and analytics requests. Janus neither segregates navigational and analytics requests nor requires workload based partitioning schemes. As one optimization, it may separate the workload to improve query performance (but it uses the same engine/data storage).

All requests are handled as transactions sent to the same storage engine. Janus uses multiple workers assigned to different cores to avoid possible update conflicts at L3 caches. It also allows tuning the number of cores assigned to each workload (e.g., at a rate 1:2 for navigational/analytics).

Our experiments show that this strategy has a negligible impact when running mixed workloads and brings Janus ahead of existing state-of-the-art graph engines indicating that emerging computer architectures could be excellent platforms for big data graph applications.

3. EXPERIMENTAL EVALUATION

We built a prototype of Janus by adding a graph operator layer on top of FOEDUS. The experiments were done on a single node: HPE Integrity Superdome X equipped with 240 cores (480 physical threads) and 12TB memory over 16 sockets, running Redhat 7.2. Superdome X is a good approximation of emerging servers in that it has hundreds of cores and a large amount of shared main memory. In the near future, emerging servers will come with non-volatile memory, but the Superdome X we used has volatile memory, so our experiments here focus on query performance rather than fault-tolerance and availability.

3.1 Experimental Setup

We used two, best-of-class graph engines for comparison. For navigational requests, we used the popular Neo4J graph database (community ed. v2.2.5, Sept. 2015). For analytics requests, we used GraphLab, a main memory analytics engine (latest open-source version as of Sept. 2015). We tuned them both following best practices for the hardware at hand according to their guidelines. In addition, we have them both installed and fully executed (e.g., caches, storage) in shared memory (/dev/shm).

In general, graph data consists of a graph topology and graph metadata (i.e., properties). Many graph engines store the graph topology in main memory and use external storage for the metadata. Janus stores both in main memory providing guaranteed fast access times for both. In our comparison, we intentionally used a data set without many properties as a way to avoid computing this extra, othogonal overhead in the results. For that, we generated data simulating a simple road network, where the vertex and edge tables have schemata: Vertex $[vid]$ and Edge $[src\_id, dst\_id, distance]$. Our data generator produces partitioned data files (both binary and text) using the sizing parameters p_x and p_y, which partition the space in the x and y direction, respectively. Each partition contains 32M vertices and each vertex has at most 20 (average 16) outgoing edges. So for example, for $p_x = p_y = 4$, we get 4 x 4 x 32M = 0.5B vertices. We report here results on the three data sets shown in Table 1.

3.2 Data Loading

We first measured data loading in all three systems. Figure 5 reports elapsed time to load data from files in shared memory (/dev/shm) to a graph engine. The figure shows loading times only for Janus and GraphLab. Neo4J was much slower and its loading times are not reported in the figure to ease the presentation of the other two systems. Indicatively, loading the small data set (vertex/edge tables/indices) to Neo4J takes 46 minutes; a significant amount of this time goes to indexing and logging. As we see, data loading in Janus is very fast and increasing the data size does not significantly affect load time unlike GraphLab, which makes a couple of passes over the data.

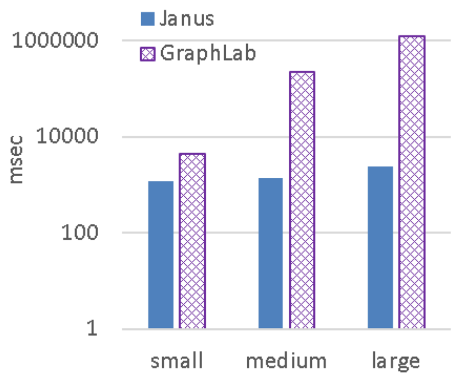


Figure 5: Data loading on GraphLab and Janus (time on y-axis in log-scale)

Next, we measured how each system processes workloads. In particular, we measured throughput (in transactions per second, TPS) for navigational requests and latency for analytics request. As the experiments show, Janus processes both navigational and analytics workloads very efficiently.

3.3 Experiments with navigational workloads

For navigational workloads, we used a randomly generated query workload of navigational queries, such as the shortest path between two vertices v_1 and v_2 placed at a small distance to each other, i.e., less than 10 hops away on average (64 at most). We compared Janus to Neo4J. GraphLab is an analytics engine and so it does not do well on this workload, as we also noticed empirically in our experiments. As we see next in Figure 7, Janus processes navigational requests quite effectively. And it is orders of magnitude faster than Neo4J. For space considerations, we omit detailed results here as the difference from Neo4J is vast. Indicatively, for the small data set, Neo4J processed on average 42 TPS, while Janus throughput is around 150,000 TPS and this difference increases a lot with the data size.

3.4 Experiments with analytics workloads

For analytics workloads, we measured latency (in milliseconds) with a representative analytics computation, namely single source shortest path (SSSP), which visits a very large portion of the graph. For fairness, both GraphLab and Janus use the same distributed Bellman-Ford algorithm for this computation [5]. Neo4J is a navigational engine and so it does not perform well on this workload, as we also verified in our experiments. Figure 6 illustrates the latency in executing SSSP on both engines for the three data sets. Janus is three orders of magnitude faster than GraphLab for the same computation over the same data. The reason is that Janus is designed from scratch as a scale-up engine for next-generation servers, whilst GraphLab does not seem to scale well and to leverage all the available computing resources.

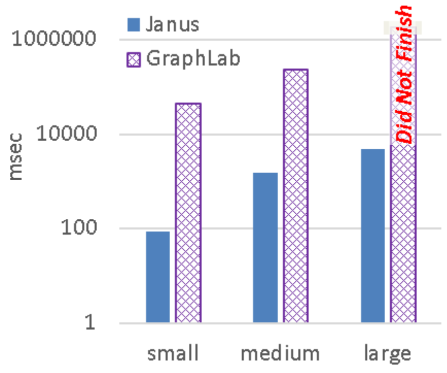


Figure 6: Analytics on GraphLab and Janus (time on y-axis in log-scale)

3.5 Experiments with mixed workloads

Next, we tested Janus with mixed, navigational and analytics workloads. Figures 7 and 8 show that Janus runs efficiently and concurrently such workloads. In addition, both figures show that the overhead in running mixed workloads (represented as the relative difference between each pair of bars) is quite small.

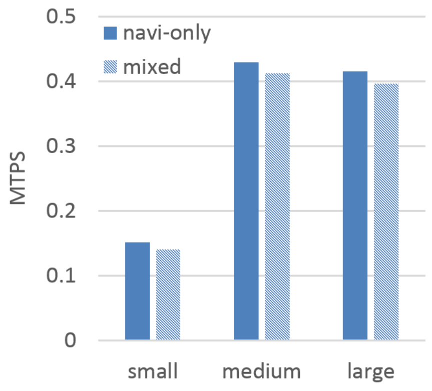


Figure 7: Navigational vs. Mixed on Janus

In particular, Figure 7 illustrates throughput in million TPS (MTPS) for a navigational only workload versus a mixed workload. The mixed workload contains the exact same navigational requests as the navigational workload and in addition, it contains a number of analytics requests to be run concurrently with the navigational ones. Throughput was expected to be constant among the various data sets because each query touches roughly the same number of vertices. However, throughput for the small data set is lower because this data set is too small, fits in one partition, and is processed by only one core.

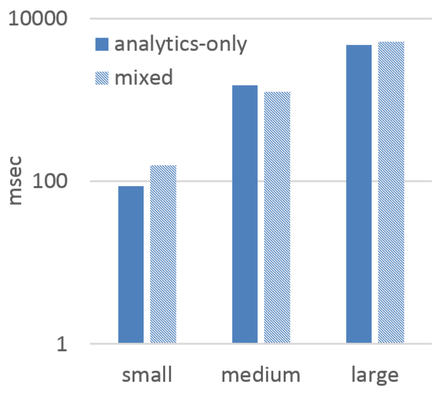


Figure 8: Analytics vs. Mixed on Janus

Figure 8 shows latency in milliseconds (msec) for an analytics only workload versus a mixed workload. The mixed workload contains the exact same analytics requests as the analytics workload and in addition, it contains a number of navigational requests to be run concurrently with the analytics ones. Observe here that Janus runs the mixed workload with very similar latency as the analytics one.

Note that for mixed workloads, as an optimization Janus can employ different numbers of cores assigned for navigational and analytics requests. In the experiments shown in the plots, we used an 1:2 core ratio for navigational, 4-cores/socket, and analytics, 8-cores/socket, for a total of 16 sockets available in our configuration.

4. RELATED WORK

State of the art graph management systems are optimized for different kinds of workloads and can be broadly classified into two categories [9]: i) navigational or online, and ii) analytics 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 graph systems include key-value stores as Neo4j [15], HypergraphDB [8], etc. and RDF stores such as Jena [3, 21] and AllegroGraph [1]. Analytics 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 include Giraph [2], GraphLab [12], Pregel [13], etc. However, most graph engines can efficiently support one workload class, either navigational or analytics. Those that claim to do both either segregate the workloads (and send them onto different engines) or maintain separate partitioning schemes, or both. Such solutions have a significant performance and computation overhead.

Many analytics engines are scale-out (e.g., GraphX [6]). Some other graph engines follow another approach for scale-up and use extremely light-weight threads as in Ligra [18] and Galois [16]. These systems are tuned only for analytics however. External memory graph processing systems like FlashGraph [22] and in-memory graph processing systems like GEMS [14] focus only on graph algorithms for large graphs. Teradata's graph engine offering with Aster [19] enables native processing of large-scale graph analytics queries, but does not support navigational, short query requests.

The distributed, in-memory graph system Trinity [17] supports low-latency online query processing and high-throughput offine analytics on large graphs. However, Trinity is tuned more for offline analytics, does not handle updates, and does not support the workloads concurrently as Janus does.

5. CONCLUSIONS

We presented Janus, a general purpose graph engine that provides excellent scale-up performance for navigational and analytics requests executing concurrently. Janus is architected for modern, many-core servers with large memory. It relies on transaction processing logic to enable real-time and consistent graph requests in the presence of concurrent data ingestion and to efficiently parallelize complex computations onto a large number of CPU cores enforcing robust concurrency control. Our first experiments show a significant performance benefit comparing to existing graph engines.

Although transaction processing can significantly improve the performance of several classes of graph queries, we do not believe that this holds for all possible graph queries. Based on our current knowledge and experience, our hypothesis is that optimization problems and algorithms, such as shortest-path, on a large graph generally have a good fit to Janus because they can leverage the benefits of serializable communications between CPU cores. Without such guarantees, the algorithms might have to repeat forever without convergence. On the other hand, probabilistic inference, such as topic modeling, might not benefit tremendously from transaction processing because both the intermediate and final results are probabilistic. Non-serializable communications might suffice in those cases.

Janus is at a prototype stage and thus, as far as future work goes, there are several possible directions to follow and features to add. An interesting problem is to optimize the latency of tiny updates (e.g., changing just a few edges) while Janus is running queries. Prior work [20] shows that FOEDUS, Janus' underlying data management engine, can efficiently handle concurrent updates and queries with frequent read-write conflicts. An additional future task is to evaluate Janus scalability against real-world graph application workloads.

We believe that next-generation computer architectures as HPE's The Machine are coming fast and graph applications could benefit tremendously from them. We are looking forward to an interesting discussion with the CIDR community. We appreciate and welcome feedback and suggestions.

6. REFERENCES

[1] AllegroGraph. *Available at: http://franz.com/agraph/allegrograph/*, 2016.

[2] Apache Giraph. *Open source project. Available at: http://giraph.apache.org/*, 2016.

[3] Apache Jena. *Open source project. Available at: https://jena.apache.org/*, 2016.

[4] F. McSherry et al. Scalability! But at what COST? In *HotOS XV*, 2015.

[5] GraphLab. *Open source project. Available at: https://dato.com/products/create/open_source.html*, 2015.

[6] GraphX. *Apache Spark's API for graphs and graph-parallel computation. Available at: http://spark.apache.org/graphx/*, 2016.

[7] Hewlett Packard Enterprise. *The Machine: A new kind of computer. Available at: http://labs.hpe.com/research/themachine/*, 2016.

[8] HypergraphDB. *Available at: http://www.hypergraphdb.org/*, 2016.

[9] A. Khan and S. Elnikety. Systems for Big-Graphs. *PVLDB*, 7(13):1709–1710, 2014.

[10] H. Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *ACM SIGMOD*, 2015.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[12] 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, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, pages 340–349, 2010.

[13] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010.

[14] 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.*, pages 139–156. 2015.

[15] Neo4J. *Available at: http://www.neo4j.org*, 2016.

[16] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471, 2013.

[17] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 505–516, 2013.

[18] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146, 2013.

[19] 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*, 7(13):1405–1416, 2014.

[20] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. In *VLDB*, 2017.

[21] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in jena2. In *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8, 2003*, pages 131–150, 2003.

[22] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 45–58, 2015.