Do We Need Specialized Graph Databases? Benchmarking Real-Time Social Networking Applications

Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{apacaci,r32zhou,jimmylin,tamer.ozsu}@uwaterloo.ca

ABSTRACT

With the advent of online social networks, there is an increasing demand for storage and processing of graph-structured data. Social networking applications pose new challenges to data management systems due to demand for real-time querying and manipulation of the graph structure. Recently, several systems specialized systems for graph-structured data have been introduced. However, whether we should abandon mature RDBMS technology for graph databases remains an ongoing discussion. In this paper we present an graph database benchmarking architecture built on the existing LDBC Social Network Benchmark. Our proposed architecture stresses the systems with an interactive transactional workload to better simulate the real-time nature of social networking applications. Using this improved architecture, we evaluated a selection of specialized graph databases, RDF stores, and RDBMSes adapted for graphs. We do not find that specialized graph databases provide definitively better performance.

1 INTRODUCTION

Graphs provide an intuitive abstraction to model objects and relationships. The highly-connected structure of many natural phenomena, such as road, biological, and social networks make graphs an obvious choice in modelling. Recently, storage and processing of such graph-structured data have attracted significant interest from both industry and academia and have led to the development of many graph analytics systems and graph databases. Graph analytics systems such as Pregel, Giraph, and PowerGraph specialize in OLAP-like batch-processing of graph-structured data on large computing clusters. Graph databases, on the other hand, focus on real-time querying and manipulation of entities, relationships, and the graph structure. Unlike RDBMSes, graph databases treat relationships as first class citizens and enable efficient traversals by native graph storage and index-free adjacency list access.

Despite recent interest in specialized graph analytics systems, some studies favour the use of RDBMSes for graph analytics for two main reasons: (i) over thirty years of research and experience in robust RDBMS technology, and (ii) its dominance in data analytic ecosystems in enterprise settings. Studies show that special-purpose graph analytics engines do not necessarily provide the best performance across all scenarios. Indeed, relational models can provide competitive performance for various graph analytic tasks, especially on single node, out-of-memory settings [5, 8].

Similar arguments can be made for OLTP-like graph workloads; however, there are no comprehensive studies of existing systems for real-world, dynamic graph workloads such as online social networks. Many studies focus on comparisons between different graph database engines and graph analytics systems [7, 10]. Although there are some studies comparing graph databases with relational models [2, 3, 6, 11], the real-time aspect of graph applications is mostly ignored and more complex graph traversals are not tested.

Our objective in this paper is twofold: (i) to propose and implement an improved graph database benchmarking architecture for real-time transaction processing and (ii) to present an experimental comparison of various graph data management solutions in online social networking scenarios. We decided to adopt the LDBC Social Network Benchmark Interactive Workload [4] due to the realistic characteristics of its generated data and queries. By integrating Kafka into the existing workload driver and ingesting updates from a Kafka queue, data ingestion and update streams can be processed in real-time to better reflect the streaming nature of updates. Using the proposed architecture, we study the performance of various data management solutions for online social networking workloads. Our study is most similar to [2] in terms of objectives; however, we believe that adopting a more realistic social graph and a streaming transactional workload better represents the characteristics of present-day social networking applications.

Our experiments show that, just as in the case of graph analytics [5], specialized graph databases do not necessarily provide the best performance across all scenarios. RDBMSes exhibits competitive performance under an interactive transactional workload in single node deployments.

The contributions of this paper can be summarized as follows:

- We implement a Kafka-based update mechanism on top of the LDBC SNB Interactive Workload to better reflect the real-time, streaming aspect of social networking applications.
- We introduce a reference implementation for the LDBC SNB Interactive Workload in the Gremlin query language that can be executed against any TinkerPop3-compliant graph database.
- We conduct an extensive performance analysis of various data modelling approaches (RDBMS, graph API over RDBMS, RDF

GRADES'17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5038-9/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3078447.3078459

store, and specialized graph database) using our proposed benchmark architecture to investigate their performance for social networking applications.

2 BACKGROUND

2.1 Graph Databases

Although traditional relational databases are sufficient to represent and process graph-structured data, they fail to provide intuitive interfaces and efficient operations for graph queries such as path queries, neighbourhood traversal, etc. Representing highly-connected data in the relational model results in a large amount of many-to-many relations, which can produce complex, join-heavy SQL statements for graph queries. By focusing not on the entities but rather the relationships among the entities, graph databases and RDF stores can offer efficient processing of graph operations like reachability queries and pattern matching. RDF stores represent graphs as collections of triples, whereas graph databases mostly adopt the adjacency list format and store adjacency information next to each entity. This enables graph databases to provide efficient graph-centric operations such as retrieving the neighbours of a vertex. Unlike in the relational model, the performance of such operations is not affected by the data size, i.e., number of vertices in the graph.

Most existing graph databases such as Neo4j, TitanDB, OrientDB employ the *property graph* model, which is a directed, edge-labeled multi-graph with an arbitrary number of key-value pairs attached to vertices and edges. Unlike RDF databases which have a structured, standardized query language called SPARQL, graph databases lack a standardized interface. Although most vendors have proprietary APIs and languages such as Neo4j's Cypher, there exist open-source efforts to unify the graph processing space; the most-prominent example being the TinkerPop project.[1]

The Apache TinkerPop3 stack provides a collection of tools and libraries for storage, querying, and analysis of graph-structured data. The Gremlin Structure API and the Gremlin query language lie at the core of TinkerPop3 stack. The former provides a common set of interfaces for the property graph model, and the latter defines a procedural query language structured around the Gremlin Structure API. They can be considered analogues to JDBC and SQL for graph databases, providing a standardized, unified way of querying and processing data modelled as a property graph.

2.2 LDBC Social Network Benchmark

The Linked Data Benchmark Council (LDBC) [1] is a joint effort from academia and industry to establish benchmarking practices for evaluating RDF and graph data management systems, similar to the Transaction Processing Performance Council (TPC).[2] The main objective of the LDBC is to specify benchmark specifications, procedures, and publish benchmarking results for different technological solutions. The LDBC Social Network Benchmark (LDBC SNB) [4] models a social network graph and introduces three different workloads on this common graph. The LDBC SNB Interactive Workload can be considered an OLTP-like workload and is designed to simulate real-world interactions in social networking applications. The

[1]http://tinkerpop.apache.org/
[2]http://www.tpc.org/

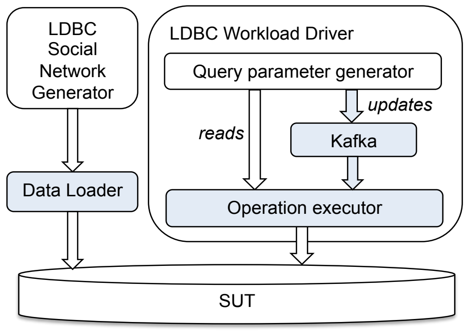


Figure 1: Our benchmarking architecture, integrating Kafka into the LDBC Workload Driver. Our contributions are highlighted.

synthetic data generator simulates the user activity of a social network for a given period of time and generates a social network graph with realistic distributions and correlations. The generated dataset contains two parts: (i) a static part which is loaded into the system as an intermediate state of the social network, (ii) updates which are played out on this intermediate state.

The LDBC SNB Interactive Workload specifies a set of read-only traversals that touch a small portion of the graph and concurrent update transactions that modify the social network's structure. The majority of the read-only traversals are simple, common social networking operations that involve transitive closure, one-hop neighbourhood, etc. The rest of the read-only traversals are complex operations that are usually beyond the functionality of real-world social network systems due to their online nature [4].

The LDBC SNB Interactive Workload uses a dependency tracking strategy to parallelize its highly-connected workload. In a nutshell, each update transaction is scheduled to be executed some time after its dependent transactions are guaranteed to be executed, e.g., a post cannot be created before the dependent user and forum are created. Such schedule-based execution enables the driver to maintain a pre-set transaction rate and test whether the system under test (SUT) can maintain operation at a steady rate. On the other hand, this execution strategy does not completely simulate a real-time workload of an online social network. Rather, the primary principle of the LDBC SNB Interactive Workload is to evaluate technical *choke points* of the SUT, that is, queries that will exhibit the behaviour of the SUT under difficult aspects of query execution and optimization [4].

3 SYSTEM ARCHITECTURE

3.1 Benchmarking Architecture

Figure 1 shows our overall benchmarking architecture, where our contributions are highlighted. First, the static portion of the synthetic social network is generated by the LDBC Data Generator, then loaded into the system under test (SUT) using vendor-specific loading mechanisms. The LDBC Workload Driver generates query parameters, which are consumed by operation executors. For read-only queries, operation executors can directly interact with the SUT, whereas updates are processed through a dedicated Kafka

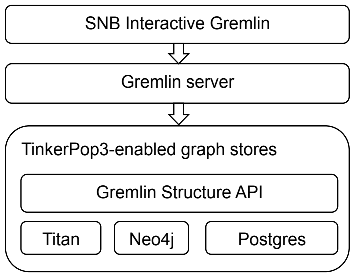


Figure 2: Execution of Gremlin queries on TinkerPop3-compliant databases via the Gremlin Server.

queue. In brief, the LDBC Workload Driver populates the Kafka queue using the generated parameters; then, update handlers read from the Kafka queue and execute updates on the SUT. We designed the Kafka integration with two considerations in mind: (i) to more accurately simulate the streaming nature of updates in online social networks and (ii) to achieve better stress testing of the SUT.

Real-world social network interactions often follow the streaming nature of updates, as seen in Twitter's GraphJet [9]. GraphJet supports concurrent read queries, while updates are handled through a single writer reading from a Kafka queue. The Kafka integration into the LDBC Workload Driver enables us to simulate a continuous stream of updates and better represent the dynamic nature of online social networks. Kafka has become the *de facto* architecture for streaming applications, used by LinkedIn,[3] Twitter,[4] and Pinterest,[5] just to name a few organizations. Thus, our benchmarking architecture better mirrors industry deployments.

Furthermore, Kafka integration achieves better stress testing of the SUT. Although the LDBC Workload Driver provides a time compression ratio parameter to achieve certain throughputs, this is accomplished by coarsening the granularity of scheduled and dependency times. This puts an upper bound on the throughput that can be achieved while maintaining correct ordering for updates. Replacing the scheduled execution with a single Kafka queue for updates guarantees correct ordering of the updates at maximum throughput.

We have open-sourced our modified driver with this Kafka integration so that the community can replicate and build on our work.[6]

3.2 Reference Implementations

We created a reference implementation of the LDBC SNB Interactive Workload in the Gremlin language as well as SQL.[7] The Gremlin implementation contains all the queries of the original benchmark specification implemented as Gremlin traversals and helper utilities for graph loading. It is used by the LDBC Workload Driver to execute the LDBC SNB Interactive Workload on TinkerPop3-compliant graph databases.

[3]https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin
[4]https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time
[5]https://medium.com/@Pinterest_Engineering/introducing-pinterest-secor-e868d9400bec
[6]https://github.com/anilpacaci/ldbc_driver/
[7]https://github.com/anilpacaci/graph-benchmarking/

The SQL implementation is formulated on a schema where vertices and edges are represented as separate tables. The reference implementation contains helper utilities to import the social network generated by the LDBC data generator into a Postgres database.

In Figure 2, we show the execution of Gremlin queries over TinkerPop3-compliant databases. Gremlin queries are submitted to the Gremlin Server, then executed over the underlying TinkerPop3 graph database.

4 EXPERIMENTAL EVALUATION

In this section, we provide experimental performance analysis of various graph data management solutions under a real-time transactional workload.

4.1 Experimental Setup

The database systems used in this study are listed below. Selected systems cover relational, graph, and RDF databases with different data models, interfaces, and query languages.

- TitanDB v1.1 with Cassandra (Titan-C) and BerkeleyDB (Titan-B) storage backends. TitanDB is an open-source, real-time graph processing layer built on various third-party storage solutions. Cassandra is run as a separate process, while BerkeleyDB is used in embedded mode. Both systems were tested using the queries implemented in the Gremlin query language.
- Neo4j v2.3.6 (Neo4j) is a single-node, specialized graph database. Being a TinkerPop-compatible graph database, Neo4j supports Gremlin in addition to its native query language Cypher. It was tested with queries implemented in both query languages.
- Virtuoso Opensource v7.2.4 (Virtuoso) is an RDBMS with column store support for RDF processing. It was used both as an RDBMS and as an RDF store, tested with native SQL and SPARQL queries, respectively.
- Postgres v9.5 (Postgres) is a popular RDBMS implemented as a row store. It was tested with native SQL queries.
- Sqlg v1.3.3 (Sqlg) is an implementation of the TinkerPop3 API on Postgres. Initially, we developed a simple implementation for the TinkerPop3 API on an RDBMS but we switched to Sqlg due to its better performance. It was tested with queries implemented in the Gremlin query language.

The graph databases (Neo4j and TitanDB) implement their own adjacency-list based custom data model on the underlying storage engine, over which user does not have control. Virtuoso-RDF employs the single table with extensive indexing approach, where the entire graph is stored in a single relational table with multiple indexes. SPARQL queries on the RDF graph are translated into SQL queries over these indexes and the base table. For relational databases (Virtuoso-RDBMS and Postgres), each vertex and edge type is represented by a separate table. For all systems, we created indexes on vertex IDs to prevent expensive linear scans on initial vertex look-ups. Although more advanced indexing schemes can be considered on an individual basis, we restricted indexes only on vertex IDs for fairness.

We used TinkerPop v3.2.3 for Gremlin Server-based experiments, the latest version available at the time of this study. The LDBC SNB Gremlin implementation (see Section 3.2) was used for all TinkerPop3-compliant systems. Similarly, we developed and used

Table 1: Dataset statistics and database sizes after data loading in GB.

Dataset	# of vertices	# of edges	Raw files	Neo4j	Titan-C	Titan-B	Postgres	Virtuoso-RDBMS	Virtuoso-RDF
SNB scale factor 3	10M	64M	3	20	6.3	29	11	2.4	8.5
SNB scale factor 10	34M	217M	10	65	18	107	36	7	25

the SQL implementation for Postgres (SQL) (see Section 3.2). For Virtuoso and Neo4j (Cypher), we modified and used the reference implementations provided by the LDBC[8] and an open-source Cypher implementation (Cypher).[9]

All systems were initially loaded with the synthetic social networks generated using the LDBC data generator to bring the social network into an intermediate state where the daily operations of a social networking application can be simulated. Two different scale factors were used to generate datasets for this study. The corresponding statistics for the generated datasets and database sizes of the selected systems are listed in the Table 1. Further information about the loading phase and the data ingestion performance of the systems are presented in Appendix A.

We conducted our experiments on two machines in a local cluster connected with 10 Gbit/s Ethernet (one for running the SUT, the other for running the benchmarking code). Each machine has 32 2.6 GHz cores and 256 GB RAM. All systems were configured to load the entire dataset into main memory before the benchmark execution in order to eliminate the overhead of disk I/O.

4.2 Read-Only Graph Queries

In this section, we evaluate the read-only performance of selected systems on four main types of graph queries: (i) point lookups, (ii) one-hop traversals, (iii) two-hop traversals, and (iv) single-pair shortest path queries. To obtain consistent results and to minimize noise, queries are executed on the static portion of the social network with no other concurrent query execution. Each query type is executed 100 times on the selected systems and execution times are recorded. Tables 2 and 3 show the average latency obtained from different systems on both scale factor 3 and 10 (the symbol '-' indicates that the system was unable to complete in a reasonable amount of time).

Tables 2 and 3 clearly show that TinkerPop3 compliance comes with an overhead. For Neo4j, the Gremlin interface introduces up to two orders of magnitude of performance degradation compared to the native Cypher interface. Similarly, Postgres (SQL) significantly outperforms Sqlg (Gremlin) even though both systems have the same underlying data model and storage engine. In general, TinkerPop3-compliant systems have the slowest read performance among all systems. These systems translate a complex graph operation into multiple small requests to the underlying system, whereas native, declarative query interfaces (Cypher, SQL, or SPARQL) enable the underlying system to generate an optimized query execution plan. Furthermore, performance differences become more significant with increasing query complexity.

Neo4j (Cypher) is the only system where there is no strict correlation between query latency and dataset size, which supports the case for adjacency-list based native graph storage. It enables index-free adjacency access where relationships of a given vertex can be accessed directly through pointers. On the other hand, the relational databases Virtuoso and Postgres (SQL) provide the lowest query latency across all query types. While Postgres (SQL) performs better in point-lookups and one-hop traversals, Virtuoso (SQL) outperforms Postgres (SQL) in more complex two-hop traversals and single-pair shortest path queries. Virtuoso's graph-aware engine and optimized transitivity support enable it to execute such complex graph queries efficiently. Virtuoso (SPARQL) has slightly lower performance due to query translation costs, even though the benchmark queries are graph queries.

4.3 Real-Time Interactive Workload

In this section, we used the modified the LDBC Interactive Workload described in Section 3.1 to simulate the real-time aspects of online social networking applications. A single writer was responsible for continuously consuming the Kafka queue and executing update transactions on the SUT. By changing the number of concurrent readers, we tested the throughput of the SUT under a stream of updates. Initially, we used the query mix defined by the LDBC SNB specification. However, TinkerPop3-compliant systems could not efficiently process large numbers of concurrent complex queries (i.e., 64 concurrent clients). Therefore, experiments reported in this section were performed using a query mix consisting of a two-hop neighbourhood based complex query and a set of short read-only queries (see Section 4.4). Figure 3 reports the read and write throughput on the scale factor 3 dataset using 32 concurrent readers.

We discovered that Titan-B suffers significant performance degradation under highly-concurrent reads and writes, which makes it unsuitable for this experiment. For the remainder of this section, we focus on those remaining systems with reasonable performance.

In general, RDBMSes with a native SQL interface provide the best performance under the real-time interactive workload. While read performance of selected systems are comparable and within a factor of four, Postgres (SQL) and Virtuoso (SQL) exhibit significantly better update performance and maintain up to an order of magnitude faster write throughput compared to their competitors. On the other hand, Gremlin-based systems have the lowest read and write throughput, which demonstrates the significant overhead incurred by the Gremlin Server. Considering that the majority of the query mix consists of lookups and neighbourhood retrievals, these results are inline with our findings in Tables 2 and 3.

The two specialized graph databases covered in this study, Titan-C (Gremlin) and Neo4j (Cypher), demonstrate comparable read performance, whereas Neo4j (Cypher) outperforms Titan-C (Gremlin) in number of writes per second. However, Neo4j's (Cypher) update performance suffers from sudden drops due to checkpointing

[8]https://github.com/ldbc/ldbc_snb_implementations
[9]https://github.com/PlatformLab/ldbc-snb-impls

Table 2: Query Latencies in ms — Scale Factor 3

System	Neo4j		Titan-C	Titan-B	Sqlg	Postgres	Virtuoso	
Query Language	Cypher	Gremlin	Gremlin	Gremlin	Gremlin	SQL	SQL	SPARQL
Point lookup	9.08	122	39	65	16.1	0.25	0.35	3
1-hop	12.82	101	240	223	34	1.4	2.15	1.23
2-hop	368	275	439	1271	2526	29	11.55	16.62
Shortest Path	21	4813	10732	13948	10243	2242	4.81	26

Table 3: Query Latencies in ms — Scale Factor 10

System	Neo4j		Titan-C	Titan-B	Sqlg	Postgres	Virtuoso	
Query Language	Cypher	Gremlin	Gremlin	Gremlin	Gremlin	SQL	SQL	SPARQL
Point lookup	11.16	177	42	236	16.9	0.32	0.41	3
1-hop	14.1	377	129	2117	43	1.62	2.22	1.71
2-hop	579	683	1570	12978	4408	46	15.92	52
Shortest Path	16	4053	17379	-	7003	3648	7.09	32

whereas Titan-C (Gremlin) can achieve a steady write throughput. The poor update performance of Titan-C can be attributed to storage and indexing abstractions introduced by TitanDB itself. Additionally, Titan-C must implement locking mechanisms to ensure uniqueness, since the underlying storage backend, Cassandra, does not provide transactional isolation. This further hinders Titan-C's update performance.

For Neo4j, native Cypher queries lead to better performance than traversals implemented in the Gremlin query language. Similar behaviour is observed in the Postgres case where Postgres (SQL) provides significantly better performance than Sqlg (Gremlin). In both Neo4j and Sqlg, Gremlin traversals are translated into multiple small requests to the underlying storage engine, thereby eliminating optimization opportunities.

Virtuoso SQL and SPARQL interfaces have similar read performance, but Virtuoso (SQL) has up to 3× better write throughput than Virtuoso (SPARQL). The difference can be attributed to higher index maintenance costs for Virtuoso (SPARQL), where multiple indexes over one big table must be maintained.

Despite similar read performance, Postgres (SQL) outperforms its closest competitor Virtuoso (SQL) by 1.6× in write performance. Considering that both systems use SQL queries over the same database schema, performance differences can be attributed to storage formats. Unlike Postgres row storage, Virtuoso employs columnar storage, which is known to suffer under transactional workloads with frequent updates.

4.4 Discussion

We encountered several roadblocks during our experimental study. First, the reference LDBC implementations contained several bugs. For Virtuoso's reference implementations, the provided SPARQL queries did not match the schema of the RDF data generated by the LDBC SNB graph generator. In addition, the SQL reference implementation did not correctly handle bi-directional edges, so a majority of queries returned either empty or incorrect results. We provided fixes to the issues we encountered in the existing reference implementations.[10, 11]

A more concerning problem we encountered relates to the performance of the Gremlin Server. As described in Section 3.1, the Gremlin Server is a layer on top of a TinkerPop3 implementation and enables multiple clients to communicate with the same database in a platform-agnostic manner. However, the Gremlin Server was unable to handle complex queries under a large number of concurrent clients. Concurrent complex queries of the LDBC SNB Interactive Workload caused the Gremlin Server to hang and eventually crash. Consequently, we modified the read query mix for Section 4.3 and omitted long running complex queries. Overall, our experience reveals the generally poor state of implementations and suggests that TinkerPop3 and specifically the Gremlin Server are not production ready.

Based on our findings, we draw the following conclusions:

- The lower performance of SPARQL compared to SQL for Virtuoso suggests that storing RDF data in a single relational table with extensive indexing is not favourable for transactional graph workloads due to index maintenance and query translation costs.
- The superior performance of Postgres (SQL) over Sqlg (Gremlin) indicates that translating graph queries into multiple small requests eliminates optimization opportunities and reduces efficiency compared to native SQL execution.
- Although TinkerPop3 presents an important vision to unify the graph processing space, the significantly lower performance of TinkerPop3-compliant systems suggests that it incurs a high overhead. Performance differences between Neo4j with Cypher and Gremlin further support the case against Gremlin.

Although results from previous work on graph database benchmarking favour graph databases for complex graph operations [2, 11], our findings show that RDBMSes can provide competitive performance under a concurrent transactional workload. We found

[10]https://github.com/anilpacaci/ldbc_snb_implementations
[11]https://github.com/anilpacaci/ldbc-snb-impls

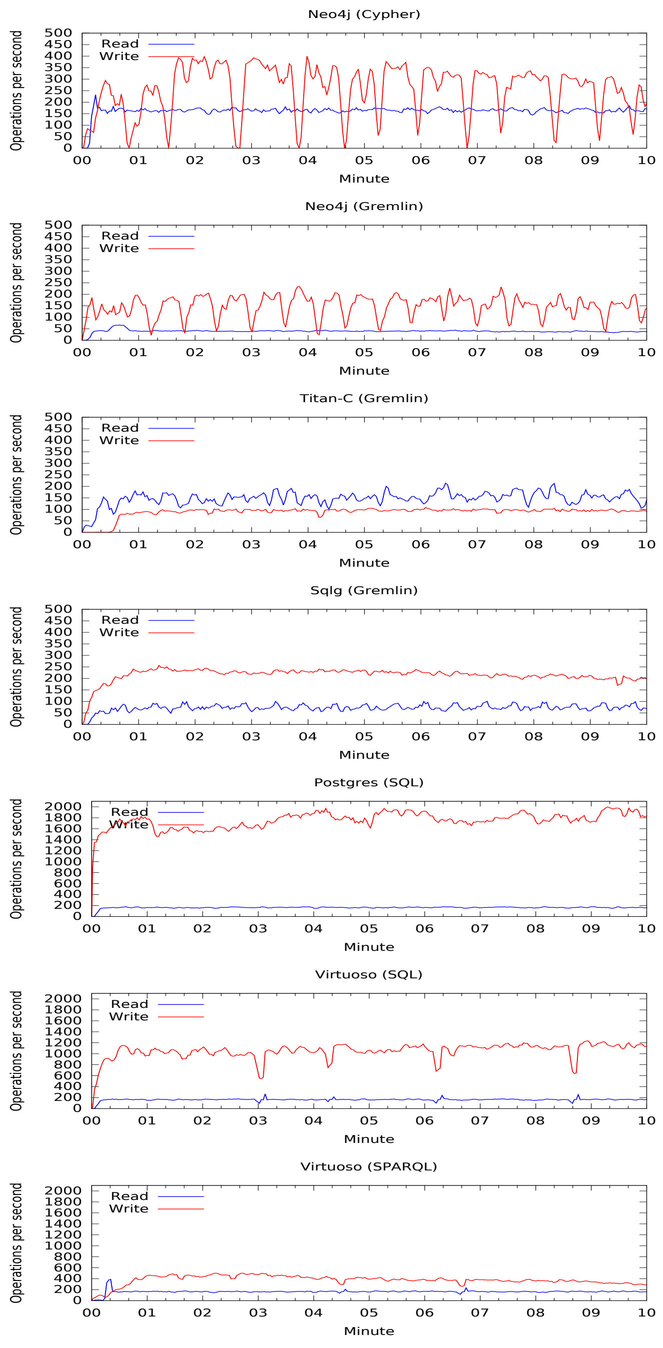


Figure 3: Aggregate read and write throughput for the real-time interactive workload.

that the relational databases Virtuoso and Postgres have low query latencies as in [4] and provides the best overall performance among all systems. Similar to [5], we believe that robust RDBMS technology can deliver competitive performance for OLTP-like online social networking applications, especially in single node settings.

5 CONCLUSIONS

In this paper, we proposed a benchmarking architecture to simulate real-time transactional workloads in social networking applications and designed an extension of the LDBC Social Network Benchmark's Interactive Workload. In addition, we developed a reference Gremlin implementation of the LDBC SNB Interactive Workload for TinkerPop3-compliant systems. Using the proposed architecture, we conducted an experimental comparison of various data modelling approaches: RDBMSes, graph API over RDBMSes, RDF stores, and specialized graph databases.

Problems we encountered during this study led us to conclude that TinkerPop3 and specifically the Gremlin Server are not ready for real-world deployments. In addition, significant performance differences of query languages in Neo4j (Cypher vs. Gremlin) and Postgres (SQL vs. Gremlin) clearly show that TinkerPop3 compatibility and the Gremlin Server integration incur significant overhead and negatively effect end-to-end system performance. Neo4j (Cypher), as a commercial system with native graph storage and optimized query execution engine, achieved higher throughput than TitanDB. Postgres provided the best overall performance, followed by Virtuoso (SQL), which led us to conclude that RDBMSes should not be ignored for interactive transactional graph workloads.

Directions for future work include considering larger datasets with different characteristics, testing the scale-out of characteristics of the systems in distributed settings, and improving our benchmarking architecture to support more realistic workloads.

ACKNOWLEDGMENTS

This research was supported by multiple Discovery Grants from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. 2014. The Linked Data Benchmark Council: a graph and RDF industry benchmarking effort. *ACM SIGMOD Record* 43, 1 (2014), 27–31.
[2] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. 2013. Benchmarking database systems for social network applications. In *GRADES*. ACM, 15.
[3] Shalini Batra and Charu Tyagi. 2012. Comparative analysis of relational and graph databases. *IJSCE* 2, 2 (2012), 509–512.
[4] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. ACM, 619–630.
[5] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. 2015. The case against specialized graph analytics engines. In *CIDR*.
[6] Vojtěch Kolomičenko, Martin Svoboda, and Irena Holubová Mlỳnková. 2013. Experimental comparison of graph databases. In *IIWAS*. ACM, 115.
[7] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. 2014. A performance evaluation of open source graph databases. In *Workshop on Parallel Programming for Analytics Applications*. ACM, 11–18.
[8] Marc Najork, Dennis Fetterly, Alan Halverson, Krishnaram Kenthapadi, and Sreenivas Gollapudi. 2012. Of hammers and nails: an empirical comparison of three paradigms for processing large graphs. In *WSDM*. ACM, 103–112.
[9] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: real-time content recommendations at Twitter. *PVLDB* 9, 13 (2016), 1281–1292.
[10] Aparna Vaikuntam and Vinodh Kumar Perumal. 2014. Evaluation of contemporary graph databases. In *ACMICC*. ACM, 6.
[11] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A comparison of a graph database and a relational database: a data provenance perspective. In *ACMSE*. ACM, 42.

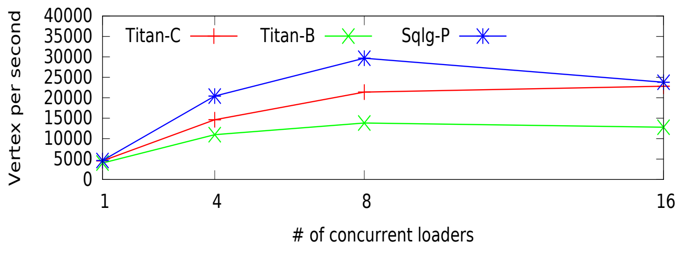
Table 4: Data loading performance for SNB scale factor 3 graph with a single loader

Dataset	Neo4j	Titan-C	Titan-B	Sqlg
Total time (min)	123	767	231	894
Vertex / second	8503	4566	4032	4660
Edge / second	10256	1465	5652	1092

A DATA INGESTION PERFORMANCE

Data loading was performed to bring the social network into an intermediate state on which social network interactions can be simulated. Initial data loading was performed using system-specific bulk loading tools for Postgres and Virtuoso, and graph-loading utilities from the LDBC Gremlin implementation for the remaining systems (see Section 3.2). System-specific bulk loading utilities enabled the loading phase to be an order of magnitude faster than that of the other systems. Table 4 reports the graph loading statistics for the SNB scale factor 3 graph for TinkerPop3-compliant systems. Sqlg, as a naive graph wrapper over a relational database, has the worst edge-insertion performance, while its vertex insertion rate is comparable to other systems. On the other hand, Neo4j's native graph storage provides the best ingestion performance in the single-loader scenario.

In addition, we evaluated the effect of concurrency on data ingestion performance. We measured the aggregate vertex and edge ingestion rates for TitanDB and Sqlg with 1 to 16 concurrent loaders. Neo4j (Gremlin) was omitted because it does not support concurrent data loading. Although TitanDB with BerkeleyDB storage provided better performance when there is a single loader, its performance degrades with increasing concurrency. In fact, TitanDB with Cassandra storage is the only system that scales with an increasing number of concurrent loaders. This can be attributed to the transactional nature of Postgres and BerkeleyDB backends. Locking introduced by transactional semantics increases latency in such write-heavy scenarios.



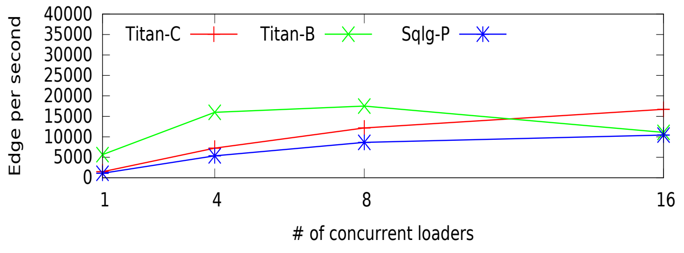


Figure 4: Aggregate data ingestion rate for varying number of concurrent loaders