Moving Recursion Out of the RDBMS for Transactional Graph Workloads

Christine F. Reilly
Computer Science Department
Skidmore College
Saratoga Springs, New York, USA
creilly@skidmore.edu

Matthew Clark
Computer Science Department
Skidmore College
Saratoga Springs, New York, USA
mclark2@skidmore.edu

***Abstract*—This paper presents work in progress that focuses on querying transactional graph data that is stored in a relational database system (RDBMS). We focus on transactional workloads where there are frequent insert and update operations. Although these types of workloads are common in social network, scientific, and business applications, much of the prior work has focused on graph analytics workloads where there is little to no change to the data over time. We introduce an approach that combines simple database queries with parallel programming, and compare our approach to the recursive SQL operations that are known to have poor performance. Our initial experiments and results provide guidance for the future directions of this project where we will refine the parallel programing approach and structure of the experiment in order to better compare these two approaches.**

***Index Terms*—relational database, recursive query, transactional graph database**

I. Introduction

Graph data is common, but finding efficient methods for storing and querying graph data remains an open research area. Our work proposes a reconsideration of the use of relational database management systems (RDBMSs) for working with graph data. Although there are a variety of graph storage systems, as discussed in Section II, there remain open questions of whether a specialized graph storage system is a better approach than a RDBMS. In this paper, we explore one aspect for improving the performance of RDBMSs with graph data: moving the recursive graph traversal operation out of the RDBMS and into a parallel program.

Graphs represent data where relationships between the data items are of importance. The nodes of the graph represent data items, and the edges represent the connections or relationships between the data items. Properties or metadata associated with nodes and edges provides additional information about the data items and relationships. Social networks are a popular application of graph data where graph nodes represent a person, a place, an event, a picture, or any other type of entity that is part of the social network. The edges of the graph represent the relationships between these entities. Figure 1 shows an example of a simple social network graph, where

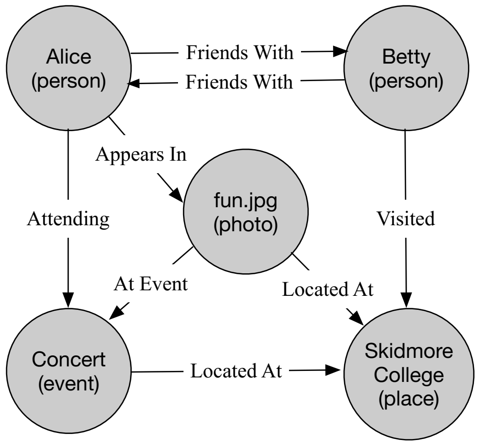

Fig. 1. Example Social Network Graph

the circles represent nodes and the labeled arrows represent the graph edges.

In addition to social networks, a wide variety of applications include data that is modeled as a graph. Tools that enable reproducibility of computational experiments capture data provenance, and this provenance data can be represented as a graph with nodes for the data and programs and edges showing the computational process [1], [2], [3]. A variety of scientific processes can be represented as graphs. Agricultural monitoring systems that utilize sensors to monitor farm activities can represent the animals, fields, and other farm entities as graph nodes and represent the interactions among these entities as graph edges [4]. Molecular dynamics, such as the processes that model protein molecules, can be represented using dynamic graphical models that are conceptually similar to the graphs that represent social networks, providing a reduction in the computational complexity required for molecular simulations [5]. Similarly, businesses have long used relational database management systems (RDBMSs) to store information about products, orders, and transactions, but modeling this business data as a graph can provide a more intuitive way to think about the data and make it easier to add in other types of data, such as product reviews [6].

Using RDBMSs for graph data has a number of benefits.

978-1-7281-9656-5/20/$31.00 ©2020 IEEE

RDBMSs are mature systems that have been developed over a span of more than 40 years on top of a large base of research. Many companies already utilize these systems for transactional data applications. Extending the use of RDBMSs to graph data applications would allow these companies to utilize their existing software for graph data and would minimize the amount of additional training that employees would need in order to work with graph data. Free and open source RDBMSs are available, which allows academic researchers to work with graph data with minimal software costs.

When using a RDBMS for graph data, recursive queries are required in order to perform graph traversals. Recursive queries have notoriously poor performance in a RDBMS. This paper presents our initial work on exploring if moving the recursion out of the RDBMS could improve graph query performance by taking better advantage of the level of parallelism available in our current computers. Sections III and IV present our experiment design and initial results. We then discuss the future directions of this research in Section V.

We focus on transactional graph data storage: the storage and querying of data in the presence of frequent insertion of new data and updates of existing data. Another design goal of this project is the use of database software that is commonly used, free, and open source. Additionally, our current research focuses is on data sizes that are large but can be served by a single machine.

II. Related Work

This section reviews related work on graph storage systems, and on data generation and benchmark workloads.

A. Graph Storage Systems

A myriad of approaches, ranging from specialized systems to RDBMSs, are currently used for storing and querying graph data. The majority of the specialized systems that have been developed for graph data storage and processing focus on graph analytics and are not suitable for transactional processing of larger data sizes where data is frequently inserted and updated. For example, the Resource Description Framework (RDF) uses a graph model to describe semantic data on the Web, and the SPARQL query language is widely used for querying RDF data [7], [8]. While there are a plethora of systems for managing RDF data, there is a lack of systems that support frequent updates to the data and scale to larger data sizes [9]. Other specialized systems have been developed for graph analytics, including distributed storage systems that can scale to larger data sizes [10], but do not support the frequent inserts and updates that are common with a transactional workload. The graph data systems that do support a transactional workload require the purchase of an enterprise license in order to support larger data sizes [11].

Our work on the use of RDBMSs for transactional graph data is motivated by related work that has demonstrated that a RDBMS performs as well as or better than a specialized graph system for graph analytics applications [12], [13]. Supporting the recursive operations that are commonly utilized for traversing graphs remains an open challenge with this approach. Prior work has explored a few directions for better supporting graph queries in RDBMSs. One option is to create new relational operations that support the types of queries that are common with graph data [14], [15]. A second option is to pair a column-store RDBMS with a graph processing system by utilizing a shared storage engine and an ehanced query optimizer [16]. The drawback to these approaches is that graph data could only be stored in the RDBMSs whose vendors have added these enhancements.

B. Data and Benchmarks

There are a number of approaches for obtaining data and benchmark workloads for graph data research.

LinkBench [17] is a data generator and database benchmark that was published by a team at Facebook but is no longer maintained [18]. We examined a currently active fork of this original code [19]. LinkBench was designed to test a single database node in the lowest layer of Facebook's storage system, which was reported to consist of many MySQL servers that provided persistent storage for the Facebook social graph. While the data and workload generators in LinkBench are designed to simulate the database activity that results from the use of an online social network, the workload does not consider the structure of a social graph.

The Linked Data Benchmark Council (LDBC) [20] specifies benchmarks and benchmark procedures for RDF and graph data. LDBC has developed a Social Network Benchmark that models a network similar to Facebook [21], [22]. This social network benchmark includes a data generator and a variety of workload generators that are currently a work-in-progress.

The Stanford Network Analysis Platform (SNAP) [23] is a general purpose network analysis and graph mining library, and the SNAP website provides a downloadable collection of network datasets. Because the SNAP software stores the graph as a file, indicating that the graph would need to fit into memory on a single computer in order for the analysis to have reasonable performance, it is not suitable for testing the larger data sizes we will explore in this research.

A variety of other approaches for graph data and workload generation are also available, and there are other academic research projects that are exploring graph data generation [24]. There are a wide variety of RDF data generation systems and datasets [9] including the Lehigh University Benchmark (LUBM) [25]. TrillionG [26] is a system that enables the generation of large synthetic graphs using a small cluster of machines.

III. Data Generation and Schema

The Linked Data Benchmark Council Social Network Benchmark data generator (LDBC-SNB Datagen) [22] was utilized to create synthetic data for our experiments. After reviewing a number of data sources and generators, as discussed in Section II, we chose the LDBC-SNB Datagen because the data mimics a realistic social network, and the generator can be

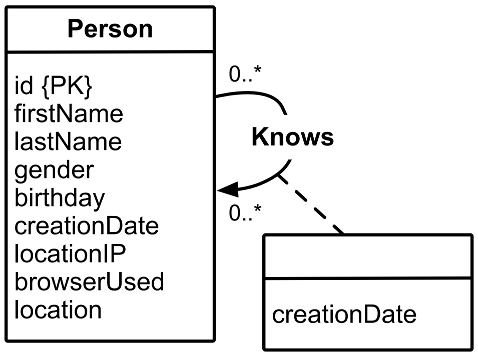

Fig. 2. Person Knows Relationship

configured to generate various sizes of data sets. Additionally, LDBC-SNB is an active research project with a code base that is regularly maintained.

We ran the data generation program in our Hadoop MapReduce cluster, then transferred the output from the Hadoop file system (HDFS) to the local disk on a workstation. The data generator is configurable to create data sets of different scales. We generated the 1GB data set for the initial experiments described in this paper and will utilize larger data sets for future experiments.

We created a relational schema for the LDBC-SNB data, following the schema description in the LDBC-SNB documentation [22]. The experiments described in this paper utilize two of the tables: the Person table and the KnowsPerson table. This portion of the schema is illustrated in Figure 2. Each row of the Person table contains information about a user in the social network. Each row of the KnowsPerson table represents a directed edge for the relationship between two people in the social network graph.

IV. Initial Experiment

The purpose of the initial experiments described in this work in progress paper is to determine if the parallel program approach is worth further exploration. The experiments are run using two machines set up as a client and a database server. Both machines have identical hardware with the exception of memory: a Xeon 4-core 3.7 Mhz processor, 2 TB hard drive, running CentOS Linux 7.6. The client machine has 32GB of RAM, while the database server machine has 128 GB of RAM. The database server is running MariaDB version 10.4 with the InnoDB storage engine.

For this initial experiment, we conduct a traversal of the KnowsPerson relationship between Person entities in the social graph, more commonly referred to as the friendship relationship. We compare the runtime of this graph traversal using two approaches: a recursive SQL query and a parallel program that issues simple database queries.

Figure 3 provides a small example to illustrate this experiment. We choose a source Person entity as the starting point in the graph, the person named Bob in Figure 3. We then traverse the graph up to a certain depth in order to find all of the Person entities that are connected to this source person. When

Listing 1. Database Schema

```
CREATE TABLE Person (
   id              BIGINT          PRIMARY KEY,
   firstName       VARCHAR(40),
   lastName        VARCHAR(40),
   gender          VARCHAR(40),
   birthday        DATE,
   creationDate    DATETIME,
   locationIP      VARCHAR(40),
   browserUsed     VARCHAR(40),
   location        BIGINT
);

CREATE TABLE KnowsPerson (
   id1             BIGINT,
   id2             BIGINT,
   creationDate    DATETIME,
   PRIMARY KEY (id1, id2),
   FOREIGN KEY (id1) REFERENCES Person(id)
       ON DELETE CASCADE,
   FOREIGN KEY (id2) REFERENCES Person(id)
       ON DELETE CASCADE
);
```

$$\begin{tikzcd}
 & & Bob \arrow[dl] \arrow[dr] & \\
 & Sue \arrow[dl] \arrow[d] \arrow[dr] & & Lin \arrow[dl] \arrow[d] \\
Eli & Mia & Jon & Jim
\end{tikzcd}$$

Fig. 3. Graph Illustrating Friendship Relationship

the maximum depth is set to 1, Bob has two connections: Sue and Lin. When the maximum depth is set to 2, Bob has 6 connections: Sue, Lin, Eli, Mia, Jon, and Jim. Notice that each connection appears exactly once in the result even when there are multiple paths from the source person to a connection, such as Jon in this example.

A. Recursive SQL

The SQL for the recursive query is shown in Listing 2. The first SQL statement sets the depth of recursion to a variable called $MAX\_ITERATIONS$. The second SQL query performs the recursive query to find all Person entities that the source person is connected to. The identifier of the source person is specified using a variable called $SOURCE\_ID$. This recursive query first seeds the recursion by selecting the source person from the Person table. The part of the query that follows the $UNION$ operator is the recursive portion: Person entities who are the destination ($id2$) in the KnowsPerson relationship will be added to the result if there is a corresponding source ($id1$) that has previously

Listing 2. SQL for Recursive Query

```
SET max_recursive_iterations
     = MAX_ITERATIONS;

WITH RECURSIVE Connections AS (
   SELECT P_source.* FROM Person P_source
   WHERE P_source.id = SOURCE_ID
   UNION
   SELECT P.* from Person P, Connections C,
      KnowsPerson K
   WHERE P.id = K.id2 AND C.id = K.id1
)
SELECT * FROM Connections
WHERE id<>SOURCE_ID;
```

Listing 3. SQL for Simple Friends Query

```
SELECT K.id2
FROM KnowsPerson K
WHERE K.id1 = SOURCE_ID;
```

been added to the Connections table. The recursive query will run for the number of iterations that was specified by the $MAX\_ITERATIONS$ variable.

B. Parallel Program

The parallel program approach moves the recursion out of the database and into the application program. The only SQL statement issued by the parallel program is a simple select query that retrieves all first level connections of a given Person entity, as shown in Listing 3. It begins by running the SQL query to find all first level friends of the source person. Then, for each of these first level friends, a parallel task is created to run the query to find the first level friends. All of the identifiers of the friends are collected into a Java ConcurrentHashMap data structure. This data structure will keep one copy of any duplicate entries. We plan to revise and refine this parallel program and will discuss these ideas in Section V.

C. Running the Experiments

We run a total of 1,001,000 instances of the experiment. The first 1000 instances are the database warm-up period and the results do not include measurements from these instances. For each experiment instance, we select a source person. The process for selecting a source person is discussed below. We then run each of the approaches for finding the Person entities that are connected to the source person. For each approach, we record the current system time as the start time, run the program to find the connections, and record the current system time after the find connections program has finished as the end time. The elapsed time (end time - start time) is written to a file along with the identifier of the source person and the number of connections that was returned by the program.

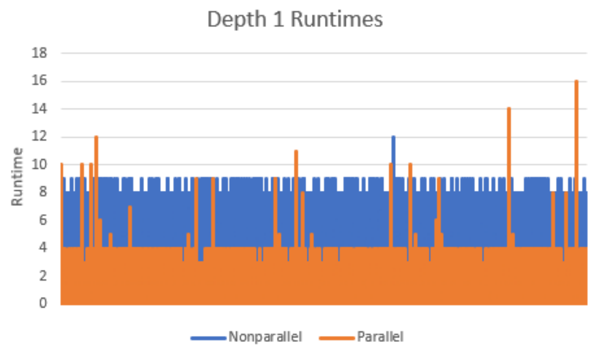

Fig. 4. Runtimes (milliseconds) with Depth of 1

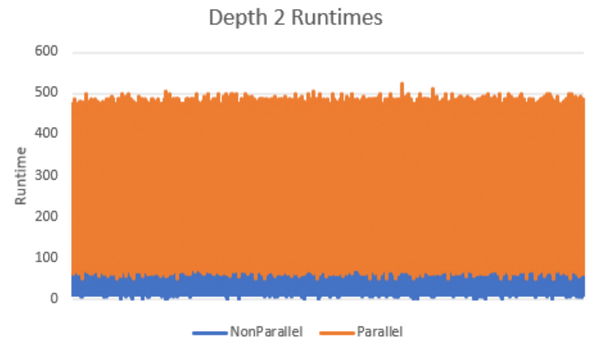

Fig. 5. Runtimes (milliseconds) with Depth of 2

In order to ensure that the source person that is selected during the experiments is one that is likely to be highly connected in the social graph, we extracted a list of the identifiers for the 100 Person entities that have the most friends and saved this list of identifiers to a file. By saving these identifiers to a file, we ensure that we will use the same set of source people for all of the experiments and that we can replicate the experiment in the future. For each experiment instance, we select a random person identifier from this set to use as the source person. We use a seed for the random number generator so that we can replicate this experiment in the future.

D. Results

We ran experiments with recursion depths of 1 and 2. The series of runtimes are shown in Figure 4 for depth 1 and Figure 5 for depth 2. The maximum, minimum, median, average, and standard deviation of the runtimes are listed in Table I for depth 1 and Table II for depth 2.

In the experiments for both depths, there were a few instances where the runtime was much longer than all other instances. We have discarded these outliers from this discussion of the results. With a depth of 1, one of the recursive query instances had a runtime of 2981 milliseconds. With a depth of 2, there were four instances with runtimes ranging from 700 to 1200 milliseconds. These four instances each base the searches around different sources. Because we have not determined a

TABLE I
RUNTIMES (MILLISECONDS) WITH DEPTH OF 1

Statistic	Recursive	Parallel
Maximum	12	16
Minimum	1	0
Median	3	1
Average	3.495	1.348
St. Dev.	1.261	0.606

TABLE II
RUNTIMES (MILLISECONDS) WITH DEPTH OF 2

Statistic	Recursive	Parallel
Maximum	345	526
Minimum	5	49
Median	51	120
Average	55.8	154.8
St. Dev.	26.4	86.0

factor related to the experiment that would impact all of these instances, we assume that these outliers were due to an external factor such as machine or network activity that is unrelated to the experiment.

For a depth of 1, the parallel programming approach had faster run times. This is likely due to the overhead of running the recursive query. If an application was only interested in retrieving the source person's connections at a depth of 1, then a simple query would be utilized instead of a recursive query.

For a depth of 2, the recursive query had faster run times than the parallel programming approach. Retrieving connections from deeper levels has not yet been tested due to refinements that need to be made to the parallel program and to the database system configurations, as discussed in Section V. These results for a depth of 2 are being used for comparison as we make these improvements.

V. FUTURE DIRECTIONS

In the results from our initial experiments, the recursive query approach had better performance than the parallel programming approach. However, we plan to continue pursuing the parallel programming approach for two reasons. First, we recognize refinements we can make to the parallel program and to the use of the database that are likely to improve the performance of the parallel program. Second, there are known limitations to the types of queries that can be supported by recursive SQL [15], and recursive SQL is generally known to have poor performance. After refining the parallel programming approach, we hypothesize that we will achieve better performance than recursive SQL at deeper levels of recursion with larger data sets and more complex queries that involve traversing the graph.

A. Parallel Program Revisions

We are planning three main revisions to the parallel program: the graph traversal algorithm, the method of parallelization, and the interactions between the parallel program and the database. Our future experiments will test various approaches to determine how each of these revisions impacts performance.

The current graph traversal algorithm, discussed in Section IV-B, uses a brute force approach where we retrieve all of the connections to the given person in each instance of the query shown in Listing 3. It is possible, perhaps likely, that a person entity is queried multiple times due to the interconnected nature of a social network. We eliminate duplicate connections at the point when the person identifiers are added to the Java ConcurrentHashMap structure. We expect that using a more refined graph traversal algorithm that records which nodes have been visited and does not re-visit nodes will lead to better performance. One approach that we will explore is implementing parallel graph traversal algorithms that others have published [27]. We will also explore utilizing the database to store the list of nodes that have already been visited during the graph traversal since the database is able to handle multiple tasks that are reading and writing the same data store.

As discussed in Section IV-B, the current program creates a parallel task for each run of the query shown in Listing 3. This could lead to a high overhead for the parallel program when there are many tasks that are each doing a very small amount of work. We will experiment with grouping parallel tasks so that there is enough parallelism for improved performance but not so much parallelism that the overhead costs dominate.

A related area where we will work to reduce overhead is in issuing queries to the database server. We will explore combining instances of the query shown in Listing 3 into batches in order to reduce the overhead of communicating with the database server, perhaps with one batch of queries per parallel task.

B. Modifications to Experiment Structure

We will structure our future experiments to better mimic the activity in a social network. Currently each instance of the experiment runs a recursive SQL query followed by a parallel program run. In the future, we will split these two approaches into separate experiments so that one experiment runs only the recursive SQL queries, and the other experiment runs only the parallel program approach. We will also modify the process for selecting the source person. Currently, we have a list of the 100 Person entities that have the highest number of first degree connections. We will modify this source person selection process so that the number of selected source person entities scales with the size of the data set. Approaches we will explore include selecting the top M-percent of person entities based on the number of first degree connections, and selecting all person entities who are connected to at least N-percent of the person entities in the graph. We will vary the values of M and N in order to determine if this setting impacts the experiment results.

After running the experiments described in this paper with the modified parallel program and experiment setup, we will progress to utilizing larger data sets and more complex queries. The LDBC-SNB data generator is capable of generating data sets at scales from a gigabyte up to a few terabytes. We will utilize a range of data set sizes in future experiments. We will also add more complex queries to the experiments by running the LDBC-SNB workload that mimics activity in a real social network.

C. Modifications to Environment

We are making adjustments to our use of MariaDB that are expected to improve query performance. In future experiments, we will use MariaDB version 10.5, which has an improved InnoDB storage engine [28], and will set the InnoDB buffer pool size to 32 GB, which is 25% of the memory on the database server. Note that the default buffer pool size of 128 MB likely resulted in slower queries for the experiments discussed in Section IV.

When we move from the current phase of initial experiments into the formal experiment phase, we will better isolate the client and server machines during the testing process. We will ensure that system maintenance, backup, and update processes are disabled during the experiment runs. We will also explore methods for isolating the two machines on the network so that other network traffic does not impact the experiment.

VI. Conclusions and Future Work

This paper described our work in progress on the comparison of recursive SQL queries in a RDBMS and the use of simple queries combined with parallel programming. Our initial results indicate a number of avenues for refinements to the parallel programming approach and to the set up of the experiment.

After completing these refinements and analyzing the final results from this project, we anticipate a variety of future directions for our research. One avenue of future work is to provide frameworks for implementing the parallel programming approach so that programmers can utilize the approach without needing an in-depth understanding of parallel programming. Another avenue of future work is to move this approach for graph traversal into distributed systems so that the data sizes can be massively scaled up.

References

[1] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Mueller, S. Castelo, C. Bautista, and J. Freire, "Your notebook is not crumby enough, REPLace it," in *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[2] Z. G. Ives, Y. Zhang, S. Han, and N. Zheng, "Dataset relationship management," in *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, USA, 2019.

[3] T. Pasquier, M. K. Lau, X. Han, E. Fong, B. S. Lerner, E. R. Boose, M. Crosas, A. M. Ellison, and M. Seltzer, "Sharing and preserving computational analyses for posterity with encapsulator," *Computing in Science Engineering*, vol. 20, no. 4, pp. 111–124, July 2018.

[4] M. Zheleva, P. Bogdanov, D.-S. Zois, W. Xiong, R. Chandra, and M. Kimball, "Smallholder agriculture in the information age: Limits and opportunities," in *Proceedings of the 2017 Workshop on Computing Within Limits*, ser. LIMITS '17. ACM, 2017, pp. 59–70.

[5] S. Olsson and F. Noé, "Dynamic graphical models of molecular kinetics," *Proceedings of the National Academy of Sciences*, 2019. [Online]. Available: https://www.pnas.org/content/early/2019/07/05/1901692116

[6] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 420–431, December 2017.

[7] W3C, "RDF 1.1 primer," https://www.w3.org/TR/rdf11-primer/, 2014.

[8] ——, "SPARQL 1.1 query language," https://www.w3.org/TR/sparql11-query/, 2013.

[9] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis, "A survey and experimental comparison of distributed SPARQL engines for very large RDF data," *Proceedings of the VLDB Endowment*, vol. 10, no. 13, pp. 2049–2060, September 2017.

[10] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica, "ZipG: A memory-efficient graph store for interactive queries," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 1149–1164. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064012

[11] "Neo4j graph platform," https://neo4j.com/.

[12] J. Fan, A. Gerald, S. Raj, and J. M. Patel, "The case against specialized graph analytics engines," in *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[13] A. Pacaci, A. Zhou, J. Lin, and M. T. Özsu, "Do we need specialized graph databases?: Benchmarking real-time social networking applications," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, ser. GRADES'17. ACM, 2017, pp. 12:1–12:7.

[14] S. Ko and W.-S. Han, "TurboGraph++: A scalable and fast graph analytics system," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. ACM, 2018, pp. 395–410.

[15] K. Zhao and J. X. Yu, "All-in-one: Graph processing in RDBMSs revisited," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 1165–1180. [Online]. Available: http://doi.acm.org/10.1145/3035918.3035943

[16] M. Paradies, W. Lehner, and C. Bornhövd, "GRAPHITE: An extensible graph traversal framework for relational database management systems," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '15. ACM, 2015, pp. 29:1–29:12.

[17] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: A database benchmark based on the Facebook social graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. ACM, 2013, pp. 1185–1196.

[18] Facebook, "Linkbench code," https://github.com/facebookarchive/linkbench.

[19] M. Callaghan, "Linkbench," https://github.com/mdcallag/linkbench.

[20] "The Linked Data Benchmark Council," http://ldbcouncil.org/ (accessed October 2020).

[21] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, "The LDBC social network benchmark: Interactive workload," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 619–630.

[22] LDBC Social Network Benchmark task force, "The LDBC social network benchmark (version 0.3.2)," Linked Data Benchmark Council, Tech. Rep., 2019. [Online]. Available: https://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf

[23] "Stanford network analysis project," http://snap.stanford.edu/.

[24] A. Prat-Pérez, J. Guisado-Gámez, X. F. Salas, P. Koupy, S. Depner, and D. B. Bartolini, "Towards a property graph generator for benchmarking," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, ser. GRADES'17, 2017, pp. 6:1–6:6.

[25] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Journal of Web Semantics*, vol. 3, no. 2-3, 2005.

[26] H. Park and M.-S. Kim, "TrillionG: A trillion-scale synthetic graph generator using a recursive vector model," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. ACM, 2017, pp. 913–928.

[27] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011, pp. 65:1–65:12.

[28] M. Mäkelä, "Faster, better, stronger, InnoDB in MariaDB Server 10.5," https://mariadb.com/resources/blog/faster-better-stronger-innodb-in-mariadb-server-10-5/.