Piecing together large puzzles, efficiently: Towards scalable loading into graph database systems

Work in progress paper

Gabriel Campero Durand
University of Magdeburg
campero@ovgu.de

Jingy Ma
University of Magdeburg
jma@st.ovgu.de

Marcus Pinnecke
University of Magdeburg
pinnecke@ovgu.de

Gunter Saake
University of Magdeburg
saake@ovgu.de

ABSTRACT

Many applications rely on network analysis to extract business intelligence from large datasets, requiring specialized graph tools such as processing frameworks (e.g. Apache Giraph, Gradoop), database systems (e.g. Neo4j, JanusGraph) or applications/libraries (e.g. NetworkX, nvGraph). A recent survey reports scalability, particularly for loading, as the foremost practical challenge faced by users. In this paper we consider the design space of tools for efficient and scalable graph bulk loading. For this we implement a prototypical loader for a property graph DBMS, using a distributed message bus. With our implementation we evaluate the impact and limits of basic optimizations. Our results confirm the expectation that bulk loading can be best supported as a server-side process. We also find, for our specific case, gains from batching writes (up to 64x speedups in our evaluation), uniform behavior across partitioning strategies, and the need for careful tuning to find the optimal configuration of batching and partitioning. In future work we aim to study loading into alternative physical storages with GeckoDB, an HTAP database system developed in our group.

Categories and Subject Descriptors

H.2.m [**Information Systems**]: Miscellaneous—*Graph-based database models*; H.2.m [**Information Systems**]: Miscellaneous—*Extraction, transformation and loading*

General Terms

Measurement

Keywords

Graph database systems, Bulk loading, Streaming graph partitioning

30th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 22.05.2018 - 25.05.2018, Wuppertal, Germany.
Copyright is held by the author/owner(s).

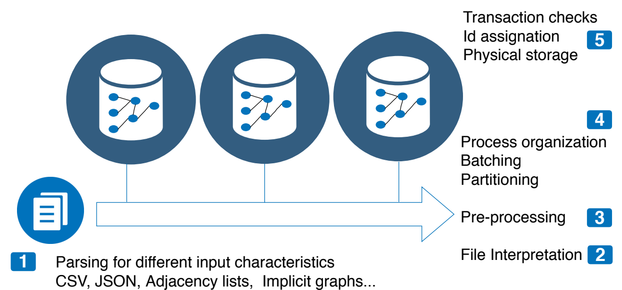

Figure 1: Bulk Loading into a Graph Storage

1. INTRODUCTION

Network analysis is one of many methods used by scientists to study large data collections. For this, data has to be represented with models based on graph theory. Namely, as a graph structure composed of nodes/vertexes and relationships/edges. Additional details of properties, labels, semantics or the direction of edges, allow to define more specific models such as RDF, hypergraphs or property graphs.

Building on such models network analysis is commonly assisted by three kinds of tools specialized for graph management:

- *Standalone applications, toolkits and libraries*, such as Gephi, NetworkX and nvGraph, provide cost-effective processing for small to medium networks in a single user environment. Accordingly they target single machine shared memory configurations. Standalone applications like Pajek and Gephi, also offer visualization.

- *Graph database systems* emphasize persistent storage and transactional guarantees in the presence of updates from a multi-user environment. Physical/logical data independence is accomplished through the adoption of high-level graph models (e.g., the property graph model), backed by different physical storage and indexing alternatives. According to their storage graph databases can be distinguished as native (i.e., with graph-specific storage) or non-native (i.e., with storage developed for other models, like documents). Example systems include Neo4j, OrientDB, ArangoDB, Gaffer, SAP Hana Graph, and JanusGraph.

- *Large scale graph processing frameworks* are characterized by their goal of supporting scale-out analytical workloads over large graphs, with failure-tolerance properties. As a result they adopt parallel and distributed strategies for batch or stream processing. Distributed job scheduling for skew handling and communication avoidance, next to I/O reduction are some of the key design characteristics of these tools. Apache Giraph, Pregel, Flink's Gelly and Gradoop are some representatives of such frameworks.

A recent survey among researchers and practitioners employing these specialized graph tools reveals several characteristics of their usage [8]. The first finding is what authors identify as *the ubiquity of large graphs*: about a third of the surveyed users report having graphs with more than 100 million edges, and more specifically, graphs on the range of billions of edges are not exclusive to large organizations but actually are used in organizations of all sizes. Second, the survey reports that graph database systems currently occupy a prominent place within the community, consitituting the most popular category of tools in use. Third, the survey finds that scalability (i.e., the need for processing efficiently larger graphs), followed by visualization and requirements for expressive query languages, are the most pressing challenges faced by users of these tools. Moreover, regarding scalability, users report that the precise challenges are inefficiencies in loading, updating and performing computations over large graphs. In this paper we share early considerations about a specific challenge of this list: bulk loading large networks into a specialized graph tool. This is a process that can become a bottleneck, and delay the time for analyis. In this sense, optimizing such process can be specially impactful, enabling analysis on data with more currency.

We organize our study as follows:

- We introduce the bulk loading process into graph tools, outlining performance-impacting aspects and usability requirements gathered by surveying the SNAP repository for network datasets (Sec. 2).
- We develop an early prototype for bulk loading into a graph database, evaluating client vs. server-side loading, request batching and different partitioning strategies (Sec. 3, 4).
- We summarize related work providing context for our research (Sec. 5).
- We conclude by suggesting takeways from our study and future work (Sec. 6).

2. THE GRAPH BULK LOADING PROCESS

Bulk loading is a process common to most graph tools (Fig. 1). We propose that the general process consists of:

1. Reading input data sources, often in tabular formats.
2. Interpreting data as nodes and edges according to rules. Intermediate data structures can be utilized.
3. Cleaning, deduplication and preprocessing (optional).
4. Process organization, where data partitioning and batch sizes can de defined.
5. Transfering this data into the physical storage model of the tool, while keeping with integrity constraints.

Intrinsic data dependencies (e.g., the fact that edges require their connected vertexes to exist) can affect how this process is organized, usually requiring several passes over the input data.

Regarding constraints, to store an edge the existence of the connected vertexes needs to be checked. For storing both vertexes and edges, integrity constraints specific to the data model might also require validation. When sources present duplicate entities, each write request might entail determining first if the entity needs to be created or not. For large graphs both checks for constraints and for duplicate entities might impact the loading time.

In terms of distributing the process, this can take place either by distributing the data sources (e.g., chunking and sharding files) or the interpreted data.

Perhaps the most pressing aspects that need to be considered by tools for the general graph data loading process we describe are *efficiency*, which for the purposes of our discussion encompasses scalability, and *usability*, which refers to fulfilling requirements from diverse input characteristics. In what follows we briefly present these aspects before advancing to the specific contributions of our study.

2.1 Performance-impacting factors

One of the main factors determining how the loading will take place, its efficiency and the possible optimizations, is the physical storage model. Paradies and Voigt [7] survey some of the more prevalent alternatives for this. Authors distinguish between choices for storing only the topology and choices for storing richer logical graph models, including labels and properties associated with nodes and edges. Among the first group they count adjacency matrixes, adjacency lists and compressed sparse rows (consisting of an ordering of edges stored in a sequential immutable array with an index of offsets to improve the access). Among the second choice they list triple tables (storing in a single table with dictionary compression the subject-object-predicate data that conform RDF triples), universal tables (wherein a single table is asssigned to edges and another to vertexes), emerging schemas (for which tables are still employed but with schemas tuned to the data), schema hashing (where item ids and properties are used as hashes to store the corresponding values in separate tables) and separate property storage (a strategy that simply separates the storage of properties from that of the topology). Specialized compressed structures, adaptive strategies and structural storage are also discussed by authors as alternative storage approaches. Finally, graph summaries like sparsifiers and spanners could be considered storage alternatives too, though specifically attuned for expected uses.

Adding to the specific storage model selected, which should reasonably determine the operations involved in bulk loading, we consider that other performance-impacting aspects pertaining to the design of an efficient bulk loading tool are: input file parsing, memory allocation, access patterns, I/O paths for persistent storage, write batching, the amount of paralellism employed and load balance, concurrency control, consensus for distributed writes, transactional management, types of cuts in data distribution, efficiency for integrity-constraint checking, and identifier assignation. These performance-impacting aspects require consideration in designing a tool for efficient graph data loading.

2.2 Usability requirements

Data loading tools should be able to assist the precise loading process of their users. This is a challenging expectation due to the diversity of data sources and formats.

In order to describe better the characteristics of data sources we consider the popular Stanford Large Network Dataset Collection [6], the SNAP repository[1]. As of the date of our evaluation, in February 2018, it consisted of 90 publicly available datasets representing social, citation, communication, collaboration and road networks, among others. The largest dataset in this collection is the Amazon Reviews dataset, consisting of approximately 36 million edges (for reviews) and around 12 GBs of compressed data. Most datasets (84) are either in CSV or TXT formats, with tab or comma separation. The remaining datasets (6, e.g. Bitcoin) also use TXT format, but following an arrangement similar to JSON. A majority of datasets (54) present the data as simple edge lists (with srcId and tgtId), which facilitate the loading. A number of datasets (9, e.g. the Ego networks) present edge data organizations that followed the idea of an adjacency list, with a single line of a file containing one source id and then several target ids. From these a small number (3, e.g. Ego-Facebook) have in addition an encoding for properties with a dictionary file and 0s and 1s to indicate if the vertex presents a given property. Another organization, which we could call implicit, is given for one dataset (e.g. Amazon Reviews). This is a specially challenging representation, as each line represents multiple edge relations.

The support for diverse input characteristics is also related to efficiency: When a tool supports a specific input source, the tool can offer optimizations related to the overall process. When a tool does not support a given input source, users can either preprocess their data to match the expected format, or, they can develop their own load process by employing operations offered by the tool. In both cases, and specially in the second, possible optimizations that the tool could perform over the complete process might be lost.

3. AN EARLY PROTOTYPE FOR LOADING INTO A GRAPH DATABASE

In order to understand the data loading process and the optimization possibilities on a general graph tool, we develop a prototype over JanusGraph, a property graph database with non-native physical storage following the schema hashing approach. This system supports Apache Cassandra, Apache HBase and Oracle Berkeley DB as storage backends. JanusGraph can be executed as a server or an application-embedded client. In both configurations JanusGraph offers a graph and a management API, in addition to maintaining socket-based read/write access to the backends, specific client-level caches and statistics.

Concretely, we propose to employ the prototype for studying the impact of server vs. client side loading, the effect of batching when loading graphs of different topologies, and distributing the edge loading process (after interpretation) through a publisher/subscriber framework to accomplish scalable loading.

4. EVALUATION

We selected JanusGraph Version 0.1.1 (May,11,2017) for our tests and Apache Cassandra 2.1.1. Our experiments were executed on a commodity multi-core machine composed of 2 Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz processors (8 cores in total) with 251 GB of memory.

[1] https://snap.stanford.edu

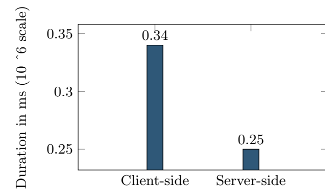

Figure 2: Client vs. Server-side Management of the Data Loading

We tackle our evaluation questions by running the data loading process on real-world datasets. We selected two datasets from different areas, with different sizes in order to make our tests more diverse. Both adopt the edge list organization. The first dataset is WikiRfA. It contains 11,402 users (voters and votes) corresponding to Requests for Adminship and votes, forming 189,004 distinct voter/candidate pairs, it is, thus a small directed, signed network. There is also a rich textual component in RfAs since each vote includes a short comment.

Wiki-RfA is an example of a real-world temporal signed network, since edges represent either positive, negative or neutral votes, and the network presents a time dimension that specifies the order in which votes are cast. In terms of topology, Wiki-RfA can be classified as a social media network, this is a kind of network similar to a social network (i.e., it can also be considered to be based on a social network), with the same scale-free properties and short paths, but that can be shaped by the affordances of the interaction platform. We choose as a second dataset the Google-Web graph, a representative of information networks and of larger datasets (800k nodes and 5M edges). In this graph, nodes represent web pages and directed edges represent hyperlinks between them.

4.1 Client vs. Server-side loading

In this section we ask, what is the right place for loading graph data, considering first if there are fundamental performance differences between carrying out the load process from client vs. server side.

As discussed previously, the loading process involves several steps according to the source files. The main steps we proposed where loading of vertexes and loading of edges; each of these involved parsing the files, creating possible inmemory mappings for ids, ordering the input items, determining the load granule (i.e., transaction size or batch size) and distributing/parallelizing the process itself. Considering that database operations can be performed as client or server codes (with the first one being passed to the systems as a series of http, websocket, language client or CLI requests, and the latter being passed as a single script, in the case of JanusGraph groovy scripts, to be executed on the server side), the first question in designing a loading tool for a graph database is to determine which of these options is the best for launching the process.

Fig. 2 presents the average time performance over 10 runs of loading data from Client/server side. We used the Wiki-Rfa dataset. The average loading time from client side is 339283.2ms (5.65 minutes). The average loading time from server side is 245320.4265ms (4.08 minutes). And the average speed up is 1.38x. From this evaluation we observe that even

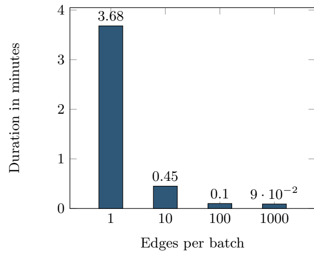

Figure 3: Effect of Batch Loading the Edges (Wiki-RfA)

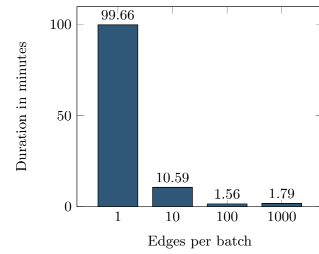

Figure 4: Effect of Batch Loading the Edges (Web-Google)

for a relatively small dataset, and without adopting any optimization, there is an evident distinction between loading in client side vs. server side, leading at least to moderate speedups.

4.2 Batch Loading

Bulk/Batch loading enables us to add large amounts of data in individual transactions. In our experiments we only considered batching alternatives for loading edges, evaluating the response time of different batch sizes.

Fig. 3 and Fig. 4 show the time taken to load all edges with different batch sizes for the different datasets. It can be seen among the two charts that batching approaches reduce the loading time significantly. The bigger the batch size, the faster the loading process. This follows a close-to-linear relationship. However, when batch sizes are increased exponentially, the loading time does not decrease in the same scale. There seems to be diminishing returns from increases in batch sizes. In fact, beyond a certain extent, the time improvement of performance from increased batch sizes becomes smaller. If the batch size is very big, it might even increase the overall time of the loading task. From our test results, the threshold of batch size where the best performance is achieved is 100.

We speculate that a possible explanation for the decreasing gains from batching could be that more data per transaction deteriorates the use of transaction caches, breaking temporal and spatial locality that appear on small transactions. A further aspect that should be considered is that large transactions could also lead to more costly distributed transactions. This was not studied here, since we did not employ multi-node backends.

One interesting thing to note is that in the dataset "Web-Google" the speedup is reduced when batch size equals 1k, while the same is not evident in Wiki-RFA. From this we can speculate that the batch size is not the only factor that affects loading performance and that topology characteristics, affecting in turn transaction cache usage, might also have an impact. Specifically, Wiki-RFA represents a more connected network than Web-Google, thus there might be more chances of reusing data already in the transaction cache, reducing loading costs. Further studies would be needed to verify these possible cases.

4.3 Partitioning

In our studies so far we have considered batching, which consisted on fitting more data inside a single transaction, in order to reduce the number of transactions employed in the loading process. In this section we consider how to organize the process with parallel transactions by partitioning the data into parallel chunks and running the loading for each chunk in separate requests to the backend. Contrasted to the previous experiments, with this approach we do not seek to reduce the number of transactions but to schedule them in such a way that some of them can be performed simultaneously, thus possibly reducing the overall runtime. To achieve this it is necessary to determine a strategy to partition the loading task. One straightforward possibility is to partition the edge set into groups of items that can be inserted separately, this is a form of partitioning over the interpreted data.

For the task of loading there is a significant difference with respect to traditional partitioning approaches. Namely that the complete graph is not available in such a way that it could enable computing a large algorithm over the graph. Instead the loading process must partition the graph with incomplete information, deciding for the location of a vertex or an edge, or a group of them, as it processes them. In spite of the limited information there is still the goal of finding a balanced partition that can also reduce communication costs during the loading process. Hence this can be defined as a streaming graph partitioning problem[9].

Authors have proposed[9] the use of different heuristics for streaming graph partitioning, such as balanced (assigning a vertex to a partition with minimal size), chunking (assuming some order in the stream, divide the stream into chunks and distribute them in a round-robin fashion), hashing items, deterministic greedy (assigning an entity to the partition where it has more items, e.g. a vertex to where it has more edges, this can be further parametrized to include penalties to large partitions), next to buffer-based ones. Authors find that these simple heuristics can bring important benefits over random cases and also reduce the edge-cuts, improving distributed graph processing[9].

We have picked 4 different partitioning strategies for our experiments [5]. These were selected due to their suitability for specifically distributing the edges, since it is not clear to us if reducing edge cuts will have or not an impact on the runtimes for our setting.

- E/E Strategy: This strategy distributes edges in a round-robin (RR) manner. It allocates many or all outgoing edges of one vertex to multiple partitions.
- V/V Strategy: The V/V strategy distributes vertexes with RR and all outgoing edges of a vertex are asigned to a single partition.
- BE Strategy: This strategy partition the graph by vertexes and meanwhile balances the amount of edges per partition. This strategy requires to sort the vertexes

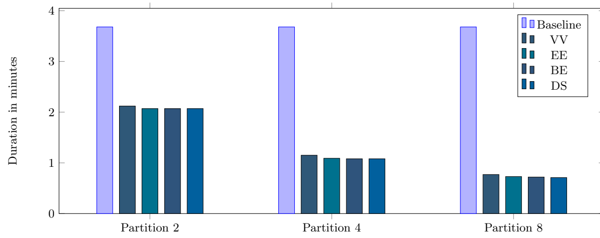

Figure 5: Loading Time using Different Partitioning Strategies (Wiki-RfA)

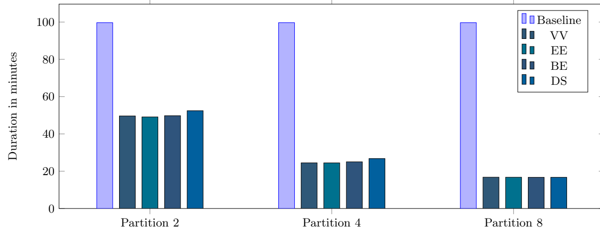

Figure 6: Loading Time Using Different Partitioning Strategies (Web-Google)

according to the number of outgoing edges in a descending order. And then it iterates over this sorted list and allocates all outgoing edges from one vertex to the currently smallest partition. It balances the edges across partitions. Thereby all outgoing edges of a vertex belong to the same partition.

- DS Strategy: This strategy basically extends the BE Strategy. It's an approximation for handling skewed data. To ease the pressure of highly connected vertexes the DS strategy allocates the edges equally across partitions. For the vertexes that have significantly more edges, this strategy separates the edges and distributes them in different partitions.

We implement the support for these partitioning by using a message passing system, Apache Kafka. When executing several JanusGraph servers (all sharing the same clustered backend), Kafka helps to organize a distributed task. For the case of loading the request is received by a single JG server. This worker is in charge of managing the load of vertexes and performing the partitioning strategies over a compact representation of the edges. Next, it sends the computed partitions to connected workers using Kafka. These in turn receive and load their partitions, following all configuration parameters given with the request (e.g., batch sizes), and reply back to the original requester via Kafka messages. Finally the original requester returns when all the partitions have been inserted.

Fig. 5 and Fig. 6 summarize the results of our evaluations with partitioning strategies. Parallel processing consistently lead to improvements over the baseline. Although parallel processing improves the performance, the overheads added due to threading and communication (i.e., more Kafka clients) limit the speedups for relatively short loading tasks. For this reason, when the loading time is relatively short (as is the case of Wiki-RfA), speedup gains decline. Thus, a careful balance is required for determining the best number of partitions according to the size of the loading task.

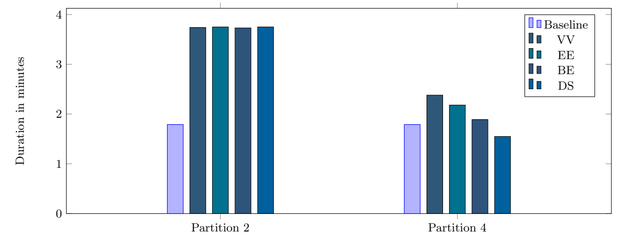

Figure 7: Loading Time Using Different Partitioning Strategies with Batch Size = 1000 (Web-Google)

We observe that loading processes using VV spent more time than using other strategies. VV is not balanced and in some situations it can lead to lower performance. In spite of the small difference for VV, we found that, overall, these basic partitioning strategies have little impact on loading time for our prototype. However we speculate that for different topologies of graph datasets, and for scaled-out architectures (where the loading is distributed but coordinated) the influence of these strategies might be different.

EE, BE and DS lead in our tests to the exact same partitions in all cases. VV leads to some small imbalances with absolute differences of 5517 and 467 for 2 partitions on Wiki-RfA and Web Google, respectively.

The combination of batching, partitioning and parallelization, as shown in Fig. 7 can actually lead to degraded performance, when the loading time is relatively short (as happens with batch sizes larger than 10). The combination was only better than the baseline for a batch size of 10, with maximum performance gains of 2x and 1.5x over all strategies for both datasets in 4 and 8 partitions respectively. Thus, the gains are sublinear. For other batch sizes, there were no improvements over the baseline. We believe that the core factor leading to this situation is that the overheads for message passing dominate the performance when the batch sizes are larger (i.e., when the tasks to perform are few). This argument is also sustained with the observation that, when not comparing against the baseline, more partitions consistently improve the performance for Google-Web, no matter the batch size, as opposed to Wiki-RfA (where the task is shorter). Regarding the differences in strategies we report one interesting case: VV for Google-Web with 2 partitions, which outperforms all cases. From our studies we know that this gain does not come from a better load balance, instead we speculate that it might be due to a good reduction in transaction commit overheads for distributed transactions, produced by the fact that the strategy assigns to a partition with a given vertex all the edges that connect to it. However further studies are needed to understand better if this is the case. For all other cases we observe mixed results regarding the strategies, and there is no clear sense of one being better than others.

5. RELATED WORK

To our knowledge there is limited related work devoted exclusively to choices for bulk loading of graph data and to improving the process.

Then et. al. [10] study optimizations at the level of DBMS design for loading a graph into an in-memory database. They propose to decompose the process into 1) Parsing (in which

the vertex data and identifiers are loaded into memory). 2) Dense vertex identification (in which, for improving memory use, vertex identifiers are sorted based on their density). 3) Relabeling, wherein in-memory dictionary encoding is adopted such that densely connected vertexes are given smaller identifiers than less connected ones. 4) Finally writing to different in-memory data structures that represent the graph (i.e. the authors consider compressed sparse rows and a map of neighbor lists).

Mainstream DBMSs like Neo4j offer useful features to improve the bulk loading process, such as loading from files bypassing the transactional layer[2], functions for batching requests and for combining writes with consistency checks via Cypher's MERGE operator.

Benchmarks like HPC-SGAB [1], Bluebench [4] and GDB [3] have tests for the loading process. The authors of GDB [3] assess the impact of batching, reporting performance gains similar to our evaluation. The authors of Bluebench [4] consider, in addition, the effect of indexing. The LDBC benchmarks study trickle updates in mixed workloads; evaluations for bulk-loading choices into graph DBMSs are, at the moment, not part of the core workloads [2].

6. CONCLUSION AND FUTURE WORK

In this paper we share early results towards designing a tool for scalable bulk loading into a graph storage. We establish the goals of our research and provide a practical evaluation using an open source database.

Stemming from our test results we can make the argument that bulk loading is better supported as a single server-side process rather than a process with intermediate operations all managed at the application/client side. Temporal structures, such as the mapping between unique identifiers and internal DBMS identifiers, can be more efficiently used when managed from server than from client side. Also, reducing the number of requests can bring performance gains by lessening the communication and interpretation costs of individual requests.

From our results we also observe batching to be a useful optimization. In our study we report a case where by moving from a batch size of 1 to 100, the loading process moves from 100 minutes to close to 1.5 minutes. Furthermore we suggest that batching should be a choice considered before others, due to its simplicity. However there are limits to this approach (too big batches might introduce large overheads on transaction failures/restarts), and performance gains do not grow in proportion to batch sizes.

Based on our results we can also conclude that parallelism is a consistently good choice, depending on the number of parallel processors available. In our study we observed that parallelization, when not combined with batching, can lead to speedups of 5.96 with 8 partitions. For partitioning we observe little to no distinction between the strategies, thus we susggest that EE could be a default strategy.

The combination of optimization alternatives: batching, partitioning, parallelization should be chosen properly, after loading tests on sample data. We have observed that using more optimizations does not necessarily translate into performance gains. In our tests with 1k batches more use of partitioning and parallelization strategies can only reduce the loading efficiency.

[2] `https://neo4j.com/blog/bulk-data-import-neo4j-3-0/`

Taken together, batching proves to be a straightforward optimization choice. It's easy to use for local settings, but for distributed scenarios parallelization becomes necessary and its combination with batching requires careful consideration and, possibly, automatically adaptive solutions.

As future work we intend to study bulk loading in different tools and evaluate the role of physical storage alternatives.

7. ACKNOWLEDGMENTS

This work was partially funded by the DFG (grant no.: SA 465/50-1).

8. REFERENCES

[1] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vanó, S. Gómez-Villamor, N. Martínez-Bazan, and J.-L. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *International Conference on Web-Age Information Management*, pages 37–48. Springer, 2010.

[2] 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*, pages 619–630. ACM, 2015.

[3] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *Social Computing (SocialCom), 2013 International Conference on*, pages 708–715. IEEE, 2013.

[4] V. Kolomičenko, M. Svoboda, and I. H. Mlỳnková. Experimental comparison of graph databases. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, page 115. ACM, 2013.

[5] A. Krause, T. Kissinger, D. Habich, H. Voigt, and W. Lehner. Partitioning strategy selection for in-memory graph pattern matching on multiprocessor systems. In *23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain*, August 2017.

[6] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[7] M. Paradies and H. Voigt. Big graph data analytics on single machines–an overview. *Datenbank-Spektrum*, 17(2):101–112, 2017.

[8] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing: A user survey. *arXiv preprint arXiv:1709.03188*, 2017.

[9] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[10] M. Then, M. Kaufmann, A. Kemper, and T. Neumann. Evaluation of parallel graph loading techniques. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 4:1–4:6, New York, NY, USA, 2016. ACM.