Benchmarking traversal operations over graph databases

Marek Ciglan [#], Alex Averbuch [*], Ladialav Hluchy [#]

[#]*Institute of Informatics, Slovak Academy of Sciences*
Bratislava, Slovakia
[*]*Swedish Institute of Computer Science*
Stockholm, Sweden
{marek.ciglan, hluchy.ui }@savba.sk
averbuch@sics.se

***Abstract*— A significant number of graph database systems has emerged in the past few years. Most aim at the management of the property graph data structure: where graph elements can be assigned with properties. In this paper, we address the need to compare the performance of different graph databases, and discuss the challenges of developing fair benchmarking methodologies. We believe that, compared to other database systems, the ability to efficiently traverse over the graph topology is unique to graph databases. As such, we focus our attention on the benchmarking of traversal operations. We describe the design of the graph traversal benchmark and present its results. The benchmark provides the means to compare the performance of different data management systems and gives us insight into the abilities and limitations of modern graph databases.**

I. Introduction

In recent years, a large number of systems for managing the graph or network data have been developed. There are numerous reasons for this, the most significant being the increased availability and importance of graph data in the form of: social, information, biological, and other types of networks. The graph data structure is a natural fit for modeling various real world phenomena, it is thus an attractive abstraction for certain data management solutions. The graph data management systems introduced in recent years can be classified in two categories – graph databases and distributed graph processing frameworks. Although the problems addressed are similar, existing systems are clearly divided and focus on one or the other. Distributed graph processing frameworks, largely inspired by Google's Pregel [1], aim to provide solutions for batch processing and mining of massive graphs that are impossible to process on a single machine, due to resource constraints. These solutions usually process the graph in memory, but different parts of graph are managed by distinct, distributed nodes. On the other hand, graph databases aim at persistent management of graph data, allowing to store and access graph data on a persistent medium. Graph database systems usually implement the property graph data model [2], where elements of a graph structure can have user defined attributes. As the graph database space is still relatively young, the different systems within it are realized using greatly varying technologies. With the increasing number of available graph database systems, it is becoming increasingly challenging for adopters to choose an appropriate solution. In this paper, we address the problem of comparing the graph database systems from the performance point of view of performing traversal operations [3] over the graph structure. We believe that the ability to traverse over the relations defined by the graph topology is the unique functionality provided by graph databases and is therefore our primary interest. There are two main contributions of the paper. First, we extend the discussion on the design of graph database benchmarks, focusing on traversal operations in a memory constrained environment, where the whole graph can not be loaded and processed in memory. Second, we present the complete design of a graph database benchmark. The paper is structured as follows: in Section II, we discuss the related work, we state the problem in Section III, we describe in more detail the design of the graph traversal benchmark in Section IV and we present the preliminary benchmark results before concluding.

II. Related work

The lack of standards in the domain of graph databases makes it difficult to compare systems. An attempt to bring a standardized interface to the diverse landscape of graph databases emerged from the open-source community and is entitled Blueprints[1]. The Blueprints project builds on the common features of graph data management systems and seeks to provide a uniform API for different systems. There is already a growing number of graph databases that, in addition to their own interface, implement the Blueprints interface as an alternative way of working with data. Concerning benchmarking, established benchmarking frameworks already exist, some partially related to graph databases, including those for: object databases, XML databases, and triple stores. However, few works specifically address the benchmarking of graph databases. An HPC benchmark for graph data processing[2] has been adopted for several graph databases by Dominguez-Sal et al. in [4]. In their subsequent work [5], authors discuss in depth the design of a benchmark suitable for graph database systems. We build on ideas proposed in their work and extend the discussion, focusing primarily on graph traversal operations.

[1]https://github.com/tinkerpop/blueprints/wiki
[2]http://www.graphanalysis.org/benchmark/index.html

III. Preliminaries and problem formulation

In this section, we first formalize the property graph data model, then discuss the requirements for the graph traversal benchmark in the rest of the section.

A. Property Graph Data Model

Informally, the property graph data model is a multigraph data structure in which graph elements (vertices and edges) can have properties/attributes. We can define the property graph as a tuple:

$$G = (V, A, P, D, L, \eta, \epsilon)$$

where V is a set of vertices, A is a multiset of directed edges (ordered pairs of vertices), P is a domain of properties, D is the domain of allowed property values for vertices, L is a domain of allowed property values for edges, $\eta : V \times P \rightarrow \mathcal{P}(D)$ is the function that maps vertices properties to their values ($\mathcal{P}(D)$ being the power set of D, meaning that a property can be associated with multiple items from D), and $\epsilon : A \times P \rightarrow L$ is the function that maps edge properties to their values.

B. Requirements for the graph traversal benchmark

Edges, defined between pairs of vertices in a graph, form the graph topology. For storing the properties of graph elements, a simple key-value store would suffice. Compared with other database options, the unique property of graph databases is the information encoded in the graph topology.

Thus, we assume a graph database should be effective at exploiting the topological information, it should provide a fast and efficient way to traverse graphs. When dealing with graph traversal problems, the usual approach is to load the graph data into the main memory and perform the traversal operation(s) while keeping the whole structure in the memory. There is a good reason for such a practice; graph traversals are characterized by their random memory access pattern and data driven computation. Although this approach poses a limit on the graph size, solutions for traversing the graph on persistent media are not common, because the cost of random accesses to a persistent medium is problematic. This is a big challenge for graph databases, where a user expects data persistence and support for fast traversal operations simultaneously – this is the main feature distinguishing graph databases from other data management systems.

We believe that a traversal benchmark should test the ability of a graph database to a) perform local traversals, starting from one or several vertices, exploring their k-hop neighbourhood and b) perform traversals of the whole graph, to support high level graph algorithms (e.g. community detection, or centrality measures). Moreover, the benchmark should test cases when the whole graph can not be cached in the main memory.

IV. Graph traversal benchmark design

This section describes the benchmark design. We begin by discussing the design decisions related to the traversal operations, the rest of the section details general benchmark features, e.g. how do we guarantee fairness and what are the properties of the data sets used.

As previously argued, efficient support for traversal operations is an important aspect of graph databases. Our intension is to measure the capabilities of graph databases to perform a) query-like traversals, where one searches for topologically related vertices for a given vertex (e.g. a breath-first traversal) b) graph analysis operations that require one or multiple traversals of the whole graph (e.g. computation of connected components, centrality measures, and community detection). We aim at studying how graph databases perform in memory constrained environments, using data sets of varying sizes. The goal of such an exercise is to test the ability of a database to deal with graphs too large to be cached in the memory.

In addition, when performing analysis algorithms, such as centrality computation, intermediate state information must be kept. We consider two cases in our benchmark. In the first case, we store the intermediate state variables in memory (this requires memory large enough to store the required state information for a given algorithm, e.g. for the HITS algorithm one keeps intermediate values for each vertex in the graph). In the second case, the state information is stored within graph elements, using the property mechanism provided by the graph database. In theory, in the latter case a user should not be worried about memory constraints as all state is managed directly by the graph database. We aim to investigate whether such an approach is feasible on currently available systems.

The primary concern of any benchmarking solution is fairness – the methodology must avoid biases. To address this problem, we have decided to adopt Blueprints as the interface to all graph databases. Our reasoning is that implementing the operations against only one interface eliminates one source of bias. Blueprints makes it possible to implement a benchmarking operation once, and for that operation to be run against different graph databases. Another concern on the fairness aspect is how to ensure the execution of exactly the same sequence of operations on the same data. For example, random generators can be used for certain operations, e.g. retrieval of k random vertices. In order to provide a fair comparison, it is necessary to use the same sequence of operations with the same input parameters on all the systems evaluated by the benchmark. In the proposed benchmark solution, we adopt the approach where we log operations and their parameters in the first run over the defined data. The logs are used it subsequent runs, where operations and their input parameters are read from the log. This allows to compare different systems fairly, as all the runs use exactly the same sequence of operations and input parameters. In addition, the logs are persistent, allowing benchmarks to be rerun on different versions of a product, and the change in performance can thus be measured.

The data used for benchmarking is also of importance. There is a difference in processing graphs with different properties; processing of a fully connected graph is more demanding than executing the same operations on a sparse graph of the same size. We believe benchmarks should be performed on data sets with properties similar to those of

real world data sets. Many network data sets (e.g. those representing social networks, the Internet, traffic networks, biological networks, and term co-occurrence networks) have small world properties. A realistic synthetic graph data set should have similar properties: power law degree distribution, small diameter, high clustering coefficient. Our aim is to use network generators with such realistic properties. We use the LFR-Benchmark generator[6], proposed by Lancichinetti and Fortunato. The generator of synthetic networks was designed primarily for testing community detection algorithms. It produces networks with power-law degree distribution and implanted communities within the network.

V. Implementation

This section discusses a few important points regarding the implementation of the benchmarking suit[3]. We have implemented it and tested on five graph databases, namely Neo4J[4], DEX[5] [7], OrientDB[6], Native RDF repository (NativeSail)[7] and a research prototype SGDB[8] [8].

As different systems take different approaches to solving the graph data management problem, their behaviors vary also. Even when using a common interface, we must deal with the small differences in underlying systems. For example, triple stores, designed primarily to handle RDF data, pose syntactic constraints on the naming of element identifiers. In addition, being edge oriented storage systems, triple stores do not provide iterators over vertices, nor support the concept of vertex properties. Furthermore, some of the benchmarked systems use the vertex identifiers provided by the user, while others generate identifiers internally. These differences have to be taken into account when implementing benchmarks.

Finally, not all of the benchmarked databases support transactions. As transaction handling requires additional processing time, systems supporting transactions are at a performance disadvantage. To limit the impact of transactions, we only commit transactions after updating 100000 graph elements.

VI. Benchmark runs

In this section we describe the setting and preliminary results of three traversal benchmarks developed in the proposed benchmarking suit. We present the results as we obtained them using the benchmarked systems in the state they were provided at the time Blueprints interface version 1.0 was released. No system specific optimizations or configurations were considered. The work on the benchmarking suite is still a work in progress. We stress that the results should be interpreted as preliminary and they might not reflect the state of the systems at a later stage.

As the loading procedure (insertion of edges into large graphs) is a time consuming process, We have decided to use a general time constraint that interrupts the loading process if it takes too long: if a block of 10000 edges takes more than sixty seconds to load. With such a slow loading procedure, it would be difficult to load large graphs.

The benchmark is designed to test the capabilities of graph databases in a memory constrained environment. The experiments where done on a low end machine with two-core 2.4GHz Intel processor and 2 gigabytes of RAM. The maximum heap size for the Java virtual machine executing the benchmarking procedure was set to 1.5 gigabytes.

The first benchmark test is aimed at the local traversal operations. The goal is to test the ability of a graph database to perform breadth-first traversals from a single vertex in a graph. The operations defined for this procedure were: a) loading of the graph definition as generated by the LFR-benchmark generator; b) for ten thousand randomly chosen vertices perform computation of local clustering coefficient (a measure expressing how clique-like is a neighbourhood of a vertex, this requires breadth first traversal two hops from the given vertex) and c) breadth first traversal for three hops from a given vertex for ten thousand randomly chosen vertices. We have performed experiments with data sets of 1000, 10000, 40000, 50000 and 100000 vertices (with mean vertex degree of 16), and with larger data sets of 200, 400, 800 thousands and 1 million vertices.

All the values are averaged over ten runs of the benchmarking procedure. The loading times are depicted in Figure 1 and the values represent the time needed to perform k edge insertions. The data is incomplete for two systems. The first one is a commercial product and we have used only the evaluation version that has a built-in limit on the number of elements that can be loaded to the system. The loading speed for the second one was too slow on the network of 100 thousand vertices (1.6 million edges) and bigger (the edge insertion procedure was slower than the defined constraint). It should be mentioned, that some of the systems have (system specific) procedures for batch loading. The batch loading procedures might be much faster than loading via standard edge insertions, but we have not used the system specific batch loading procedures for the sake of generality. Figure 2 depicts the performance of the breath-first search operations (3 hops from the given vertex) for a set of 10 thousand vertices. We can conclude that most of the benchmarked systems had stable performance for this operation even with increasing size of the network. The very same characteristic was found for the local clustering coefficient computation (we omit the figure from this paper).

The same experiments were performed for larger networks, however in our benchmark setting most systems were not able to load graphs with more than 400000 vertices because of the load time constraint. Nevertheless, the observation from the partial results was similar as in the case of smaller networks. The compute time of local traversal operations was increasing, but it was increasing sub-linearly to the network size.

The next benchmark targets global traversals with the intermediate results kept in memory. The goal is to test traversal of

[3]http://ups.savba.sk/~marek/gbench.html
[4]http://neo4j.org/
[5]http://www.sparsity-technologies.com/dex
[6]http://www.orientechnologies.com/orient-db.htm
[7]http://www.openrdf.org
[8]http://ups.savba.sk/~marek/sgdb.html

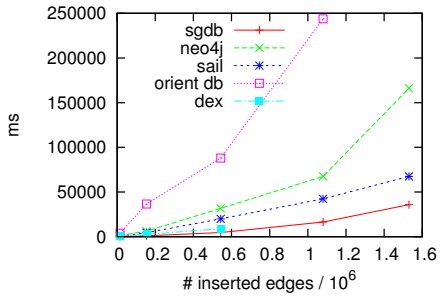

Fig. 1. Load times for the small datasets, x-axis denotes the number of edges divided by 10^6, y-axis denotes the time in milliseconds required to insert the structure.

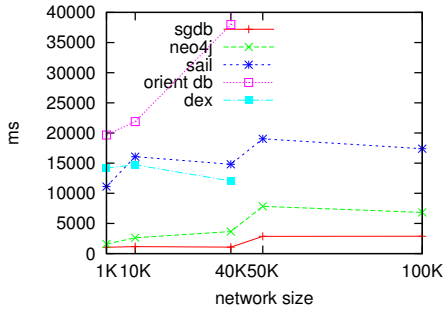

Fig. 2. Computation of breath-firest search 3 hops from a vertex for 10000 randomly chosen vertices.

the whole graph. For this purpose we have used the detection of connected components, using expansion following outgoing edges in one case and incoming edges in the second case. We have done the tests on smaller networks, in order to avoid loading problems noted in previous benchmark. The results are depicted in Figure 3 and represent the average of 10 runs of the benchmark where three systems were compared. In general, the runtime increases significantly with the increasing complexity of the network. An interesting observation is that the runtime using outgoing edges was significantly higher for two of the tested systems than the same procedure only using incoming edges.

We also tested computing connected components with intermediate data stored in the processed graph as properties. The runtime for the tested systems were an order of magnitude higher than in the case of storing intermediate data in memory, thus we do not consider this approach to be a viable option for the processing.

VII. Conclusion

We discussed the problem of benchmarking graph database systems, focusing in particular on graph traversal operations. We have proposed to study the ability of graph databases to execute traversal operations in a memory constrained environment and we have presented the design of a fair graph traversal benchmark. The work is still a work in progress; we have reported preliminary results of benchmark runs. An important

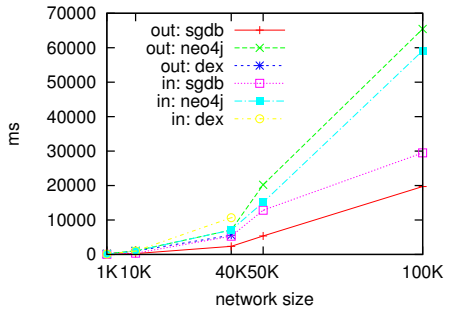

Fig. 3. Computation of connected components using in- and out-going edges.

observation is that procedures requiring the traversal of the whole graph are feasible when the structure can be cashed in memory. The operations aiming at local traversals in the vicinity of a given vertex are responsive and, even though the runtime increases with increased network size, the increase is sub-linear. The conclusion of the preliminary results is that the use cases with operations requiring local traversals in a large network are more suitable for the tested systems than those requiring traversals of the whole graph structure.

VIII. Acknowledgements

Although he was not directly involved in this work, we would like to thank Martin Neumann for the many contributions he made to parts of the software in our benchmarking suite. This work is supported by projects SMART II ITMS: 26240120029, VEGA 2/0184/10, VEGA No. 2/0054/12.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[2] M. A. Rodriguez and P. Neubauer, "Constructions from dots and lines," *Bulletin of the American Society for Information Science and Technology*, vol. 36, no. 6, pp. 35–41, August 2010.

[3] ——, "The Graph Traversal Pattern," *Graph Data Management: Techniques and Applications*, August 2011.

[4] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey, "Survey of graph database performance on the hpc scalable graph analysis benchmark," in *Proceedings of the 2010 international conference on Web-age information management*, ser. WAIM'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 37–48. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927585.1927590

[5] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, and J. L. Larriba-Pay, "A discussion on the design of graph database benchmarks," in *Proceedings of the Second TPC technology conference on Performance evaluation, measurement and characterization of complex systems*, ser. TPCTC'10. Berlin, Heidelberg: Springer-Verlag, 2011.

[6] A. Lancichinetti and S. Fortunato, "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities," *Phys. Rev. E*, vol. 80, Jul 2009.

[7] N. Martínez-Bazan, S. Gómez-Villamor, and F. Escale-Claveras, "DEX: A high-performance graph database management system," in *GDM 2011, ICDE Workshops*, 2011.

[8] M. Ciglan and K. Nørvåg, "SGDB - Simple Graph Database Optimized for Activation Spreading Computation," in *GDM 2010, DASFAA Workshops*, 2010.