EVALUATION OF DISTRIBUTED GRAPH DATABASES

Berke Egeli
Supervised by *Maciej Besta*

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Previous works on bechmarking graph databases focus on relatively small scales of dozens of cores. In this work, we measure and compare the performance of open source graph database management systems, JanusGraph and Neo4j, against Graph Database Interface (GDI) in high performance computing settings, scaling up to thousands of cores and dozens of nodes. We show that GDI has 10 times higher throughput than JanusGraph and thousands of times higher throughput than Neo4j.

1. INTRODUCTION

Motivation. Graph databases have significant relevance both in academia and industry due to their data analytics use cases [1, 2, 3]. They are designed to work with irregular data sets which can be modelled as Labeled Property Graphs or RDF graphs. Complex data can also be attached to edges and/or vertices as well in the form of rich labels and key-value properties.

Current graph databases have been tested on scales of up to dozens of compute nodes [1, 2]. Scalable Parallel Computing Lab at ETH Zürich has been developing Graph Database Interface (GDI) with the intention of scaling graph databases to millions of cores. The system is designed for high performance computing environments with very fast Remote Direct Memory Access (RDMA) networks. In order to measure the efficacy of the system there is a need to benchmark the performance of GDI against other graph databases intended to be used in distributed environments.

Contribution. Benchmarking graph databases is a challenging task at very large scales. Efficient bulk loading of data and indexing of very large graphs, configuring the database management systems to properly bulk load data and commnuicate with other nodes in the clusters, and automating the start up of very large clusters are all important aspects of benchmarking. Lack of built-in tooling to handle data loading/indexing and general scalability limitations of Neo4j in general make it a non-trivial task to measure their performance at large scales.

In this paper, we present a comparison of Linkbench query latencies, weak and strong scaling results of commercial graph database systems like JanusGraph and Neo4j agains GDI at large scales. GDI has multiple orders of magnitude higher throughput than both JanusGraph and Neo4j. GDI is also significantly more scalable than Neo4j in terms of the number of compute nodes/cores it can scale to.

Related work. Multiple surveys of graph databases (GDB) have been published over the years [1, 2, 3, 4]. Some surveys like [3] and [4] focus on mostly qualitative analysis and use cases of graph databases. Fernandes, for example, analyses popular graph databases based on their features such as "flexible schema, query language, sharding and scalability" [3] which will enable the GDBs to serve as complete and effective applications.

Monteiro [4] takes a more methodical yet still mostly qualitative approach to assessing open source GDBs like Neo4j and JanusGraph using a method called OSSpal [5], which is a methodology used to help organizations discover and evaluate high quality, free and open source systems.

There are more quantitative analysis of existing GDBs as well [1, 2]. Lissandrini et al. argued that most of the available assessments of GDBs are limited in providing experimental data regarding the performance of systems or in the cases where they provide experimental data they are incomplete in the number of systems or the set of operations they evaluate [1]. Therefore, they have developed an extensive micro- benchmarking suite which analyses 7 different GDB systems over 7 classes of queries. Their benchmarking suite, however, has been designed for a relatively small machine with 24 cores, 128 GB memory and 2 TB disk space.

Dominguez-Sal et al. extend the quantitative analysis of graph databases to high performance computing

environments [2]. They measure the performance of GDBs against graph loading, short navigations or full graph traversal operations based on the HPC Scalable Graph Analysis Benchmark [6].

In our paper, we measure the performance of graph databases with respect to a variety of query types like [1] in a high performance computing environment. The scale of the machines and the graphs we use are much larger than previous two papers at around 100x more cores and 30x larger graphs [1, 6].

2. BACKGROUND

2.1. JanusGraph

JanusGraph. JanusGraph [7] is a hybrid graph database management system. It relies on third party storage systems to act as the storage layer. By default, JanusGraph has adapters for Apache Cassandra, Apache HBase and Oracle Berkeley DB Java Edition [4, 7]. It also supports Elasticsearch, Apache Solr and Apache Lucene for indexing. JanusGraph stores graphs as a collection of adjacency list. To support this model, JanusGraph creates a new row for each node in the underlying storage system. A column is created for each attribute and edge of the node [1]. Because of this design, JanusGraph needs to access node ID index first for edge traversals. JanusGraph supports Gremlin query language and uses Apache Tinkerpop as the graph computing layer to support Gremlin [1, 7].

Cassandra. Apache Cassandra [8] is an open source, NoSQL, wide column store. It has linear scalability and high fault tolerance. Cassandra has a peer to peer architecture. The lack of master-slave architecture eliminates read/write bottlenecks. Because of these properties, it is also continuously available with no single point of failure [7].

Cassandra is an ideal backend storage system due to its high read-write throughput and linear scalability, capable of supporting large graphs.

2.2. Neo4j

Neo4j is a native graph database management system [1, 3]. Unlike JanusGraph, Neo4j has a dedicated storage system designed to support graph operations. Neo4j maintains separate files for vertices, edges, labels & types and attributes to speed up traversal operations [1]. It is described as a transactional database [3].

2.3. LinkBench

LinkBench is a database benchmark developed to measure the performance of systems with workloads similar to Facebook's production MySQL deployment which stored its social graph data [9]. Therefore, LinkBench is also a benchmark suitable for measuring the performance of graph database management systems. LinkBench benchmark measures the performance of read, insert, update and delete operations of graph nodes [9]. For edges, it measures the performance of count, range, multiget, insert, delete and update operations [9].

2.4. Graph500 Benchmarking

Graph500 is a ranking of supercomputers. It has benchmarking specifications on how to create graphs and which kernels to use in tandem with these graphs to measure the performance of the high performance computing systems. Graph500 benchmarks contain a Kronecker graph generator to create an edge list of graphs [10].

The edge tuples are modeled as (StartVertex, EndVertex, Weight). Benchmark only takes one parameter as its input which is called scale. Scale is the logarithm base two of the number of vertices. Benchmark defines a constant called the edgefactor which is the ratio of the edge count against vertex count in the graph. Edgefactor is set to 16. The number of vertices in the graph, N, is $N = 2^{scale}$. The number of edges in the graph, M, is equal to $M = 16 * N$.

3. BENCHMARK IMPLEMENTATION

JanusGraph can be used in two ways. First one is to use a cluster of servers as JanusGraph engine. Second way is embedding JanusGraph as a library inside the application code. We use Java API of JanusGraph to embed it as a library and directly connect to backend storage. The benchmark is developed in Java 11 and the JanusGraph core driver is version 0.6.2.

The benchmark receives a couple of parameters. The first and most important parameter is the path to the JanusGraph configuration file. It contains the hostnames of the storage backend nodes, caching and indexing parameters among many other configuration options which will be used to set up the JanusGraph engine and connect to the backend storage nodes. Users can also specify the number of threads a single benchmark node should start and the number of queries to execute per thread. It is also possible to set the query percentages for different types of query the system will execute. Finally, the user can give a flag to the benchmark to measure the latency of individual queries or throughput of the system.

3.1. Execution Controller

Execution controller is the module which creates and manages threads. Based on the parameters the user gives, execution controller creates a mixture of queries to be executed and assigns them to threads. Threads execute queries assigned to them and return the execution metadata to the execution controller.

Latency Mode. In the latency mode, the execution controller extracts the latency of each query and writes them to a file with the corresponding query id. There is a record in the output file for each query executed. This data is later used to build a latency histogram of each query type.

Throughput Mode. In this mode, the execution controller aggregates the total latency of queries executed by a single thread. It writes the the number of executed & failed queries and the total execution time of all queries in the a thread. Throughput, T, is later calculated as $T = (total\_\#\_of\_queries) * (max(latency\_of\_thread))$.

3.2. Queries

The queries selected for benchmarking the performance of the system adheres to LinkBench benchmark.

Insert Operations. We have two types of insert queries. The first one is for inserting vertices and the second one is for inserting edges. For the former, we create a vertex with random properties and then insert it into the graph. The latter selects two random vertices from the graph and inserts an edge between them. To insert a vertex into the graph, we use has and property steps in the Gremlin query. To insert an edge we use has, addE and to steps.

Read Operations. We have a read query for vertices. We access the vertices through their custom identifiers appended as properties. Therefore, we use a has step to read vertices.

Update Operations. Update queries are only applied to vertices. We select a random vertex and update one of its properties with a random value. We use has and property steps in the Gremlin query.

Delete Operations. Delete query is only applied to vertices and its properties. We use the inbuilt remove operation provided by the Java API to delete vertices.

Traversal Operations. We have two types of traversal operations for edges. First operation is edge count operation where we count the number of a randomly selected edge type that come out of a random vertex. We use has and count steps for this query.

The second operation is a range query where we iterate to all of the neighboring vertices from a random vertex if the random vertex and the neighbor are connected by a randomly selected edge type.

4. EXPERIMENTAL RESULTS

Experimental Setup. We run our experiments on Piz Daint, a hybrid Cray XC40/XC50 system. All of our experiments are conducted on Cray XC40 nodes, using up to 64 nodes to host graph databases and an additional 64 nodes to run benchmarking code. Each node has two Intel® Xeon® E5-2695 v4 @ 2.10GHz (2 x 18 cores, 64/128 GB RAM). We benchmark JanusGraph version 0.6.2. We use Cassandra version 4.0.4 as JanusGraph backend. For benchmarking Neo4j, we use version 4.4.4. The graphs we use in our benchmarks are generated according to the Graph500 benchmarking specifactions. Graph scales range from 21 to 25.

4.1. JanusGraph

Weak Scaling. In this experiment, we study the performance impact of varying the graph sizes while we increase the number of queries we execute in the systems. The database with the smallest graph runs on a single compute node. The smallest graph has around 2 million nodes and 33 million edges. We scale the nodes and the graphs exponentially, up to 16 compute nodes, 33 million nodes and 540 million edges. We allocate separate compute nodes to run the queries in order avoid interference between the database operations and benchmarking code. The number of database nodes and benchmark nodes are equal. Each benchmark node has 36 threads running which execute 10 thousand queries. We have a read heavy query mix with 70% read queries and 30% write queries.

In the weak scaling scenario, both systems show similar trends in terms of the increase in their throughputs and failed query percentages. As you can see from figure 1, in JanusGraph, as we double the number of compute nodes, the throughput increases by around 1.5x with each doubling from around 15 thousand queries per second with a single compute node to 150 thousand queries per second with 16 compute nodes. GDI throughput increases by around 2x with each doubling from 160 thousand queries with 2 compute nodes to 900 thousand queries with 16 compute nodes. Throughput of GDI is between 5x to 6x higher than JanusGraph. In GDI, the throughput is higher with a single compute node, whereas in JanusGraph the throughput consistently increases as the number of nodes in the system increases. We assume the throughput of GDI is higher when there is a single compute node because communication and coordination between the database nodes are eliminated.

In both systems, the percentage of failed queries stays relatively constant at around 0.05% for JanusGraph and 0.15% for GDI. This is because, as we increase the number of queries, the size of the graphs increase as well, reducing the number of query failures caused by trying to access already deleted nodes.

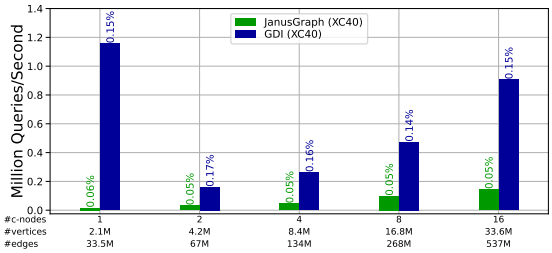

Fig. 1: *Weak scaling*: Comparison of JanusGraph and GDI throughput with different graph sizes.

Strong Scaling. In this experiment we study the performance impact of increasing the number of queries we execute while keeping the size of the graph constant. The graph has 8 million nodes and 130 million edges. The number of database and benchmark nodes range between 8 and 64 compute nodes, with increments of 8 nodes per data point. Similar to weak scaling experiment, each benchmark node has 36 threads running which execute 10 thousand queries each. The query mix is the same as the weak scaling experiment.

The throughput of both JanusGraph and GDI scales linearly as the number of database and benchmark nodes increase. As it can be seen from figure 2 GDI achieves a throughput of 4.8 million queries per second at 64 compute nodes whereas JanusGraph achieves 480 thousand queries per second, so, GDI has an order of magnitude higher throughput than JanusGraph. In both systems the throughput increases by around 6x times when the compute nodes increase from 8 nodes to 64 nodes.

As the size of the graph is fixed in all of the strong scaling experiment runs, the percentage of failed queries scale linearly as well for both of the systems. The percentage of failed queries are roughly the same for both systems, starting at around 0.1% for 8 nodes and ending at around 0.8% for 64 nodes with 0.1% increments in the successive data points.

Latency. Next, we look at the latencies of individual queries executed during the strong scaling experiment. As it can be seen from the figures 3 and 4, our code executes Linkbench queries to measure the performance of graph databases.

Figure 3 shows the latencies of individual queries on JanusGraph when we exponentially increase the number of compute nodes from 1 to 8. In each query subplot, the query latencies more or less completely overlap. This reinforces the conclusions we reached in the previous two experiments. JanusGraph performance scales linearly as the amount of compute resources and the number of concurrent query executions increase. As a result, the latency of queries stay the same as the number of nodes and queries increase. All the queries with the exception of deleting nodes has a median latency of around 1.5-2ms. For deleting nodes the median latency around 3ms.

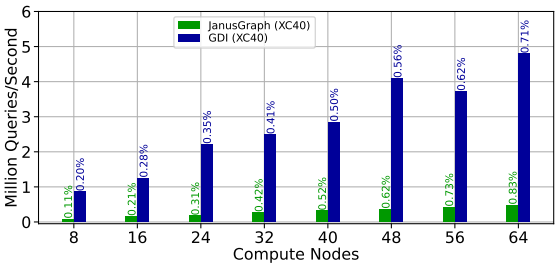

Fig. 2: *Strong Scaling*: Comparison of JanusGraph and GDI throughput with the same graph size.

Figure 4 compares the query latencies of JanusGraph and GDI. In this experiment, we used only a single database node and a benchmark node. Again, the latency results reinforce the results in the previous two experiments. The query latencies of GDI are around an order of magnitude lower than JanusGraph. The median latencies for GDI queries are around 30-50us, with the exception of delete nodes query where the median latency is around 100us.

4.2. Neo4j

Weak Scaling. Similar to JanusGraph, this experiment aims to study the performance impact of varying the graph sizes while we increase the number of queries in the system. The number of database and benchmark nodes start from 2, unlike the JanusGraph experiments, as Neo4j cluster requires at least two nodes to start up. The number of nodes go all the way up to 16, increasing exponentially. Graph sizes range from 4 million nodes and 67 million edges to 33 million nodes and 540 million edges, similar to JanusGraph weak scaling experiment. Again, each benchmark node runs 36 threads but for these experiments, unlike JanusGraph, each thread executes 1000 queries due to Neo4j scalability and performance limitations. For Neo4j clusters we have two types of database nodes, core nodes and replica nodes. Core nodes have a weight of 3 while replica nodes have a weight of 2. This means that for every 3 core nodes our clusters have 2 replica nodes. When the number of compute nodes in the cluster is not divisible by 5 we

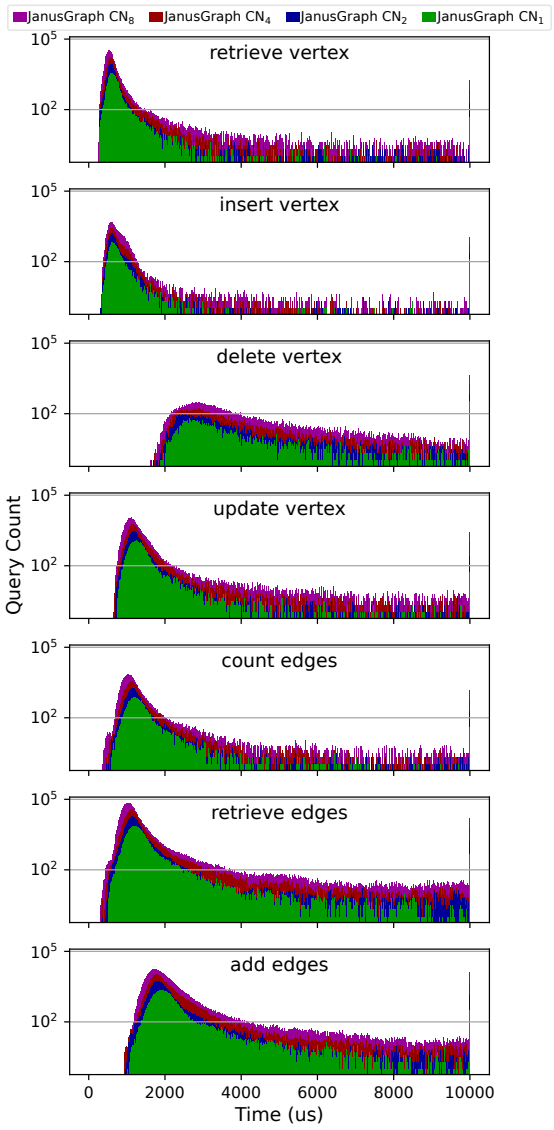

Fig. 3: *Latency*: Comparison of JanusGraph query latencies with different number of cluster nodes.

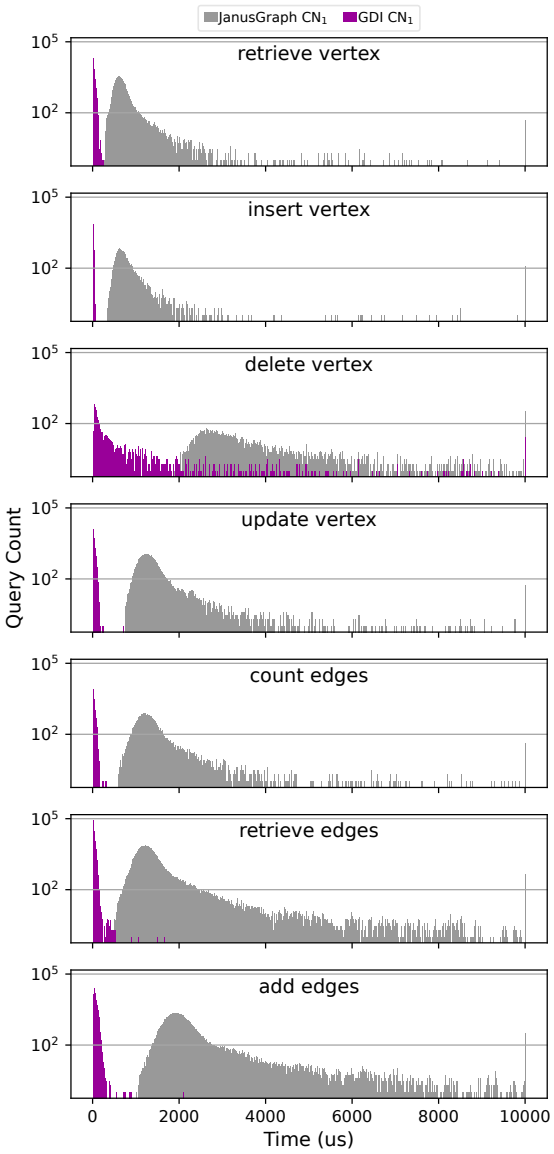

Fig. 4: *Latency*: Comparison of JanusGraph and GDI query latencies with a single node.

round the number of core nodes up and the number of replica nodes down.

Figure 5 shows that, Neo4j has 3 orders of magnitude lower throughput than GDI. The throughput of Neo4j for 2 compute nodes is around 90 queries per second while the the throughput for 8 compute nodes is around 50 queries per second, meaning that the throughput decreased by around 45% as the compute resources increased by 4 times. These results suggest to us that the throughput of Neo4j does not scale linearly as the number of database nodes in the cluster increase and that Neo4j struggles to scale horizontally. This conclusion is further supported by the fact that, as the number of compute nodes increase to 16, the percentage of failed queries drastically increase from 1% to 15%.

Similar to the weak scaling experiment of JanusGraph, the percentage of failed queries stays constant up to 8 compute nodes.

Strong Scaling. This experiment is similar to the strong scaling experiment of JanusGraph. We use the same graph with 8 million nodes and 67 million edges. Due to Neo4j's scalability issues, we increase the compute nodes from 2 to 16 with 2 compute node increments per data point. Benchmark node and the core/replica node configurations are the same as the weak scaling experiment of Neo4j.

As it can be seen from figure 6, the throughput of

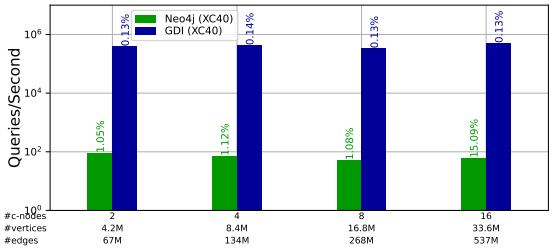

Fig. 5: *Weak scaling*: Comparison of Neo4j and GDI with different graph sizes.

Neo4j stays flat when the number of compute nodes and queries increase. The reasoning is the same as the conclusion we draw in the weak scaling experiment which is the performance of Neo4j failed to scale linearly with the number of compute nodes in the system.

In figure 6 we can see that the failed query percentages of Neo4j stay relatively constant until 14 compute nodes, unlike its counterpart in the strong scaling experiments of JanusGraph. It turns out that the benchmarking code for Neo4j copies the complete graph dataset into each database node for Neo4j unlike JanusGraph and GDI which split the data equally among all of the database nodes. This prevents the percentages of failed queries rising with the rise in the number of queries.

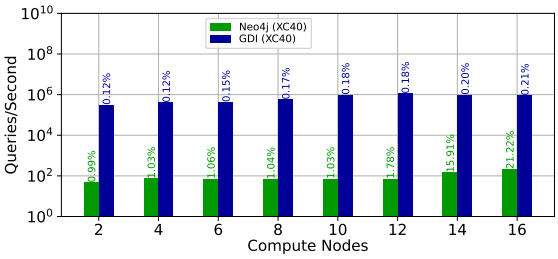

Fig. 6: *Strong Scaling*: Comparison of Neo4j and GDI with the same graph size.

Latency. We turn our attention to the latency of Linkbench queries on Neo4j and compare it to GDI. In figure 7, we scale the number of compute nodes from 2 to 8. As opposed to JanusGraph, we ommitted the latency results of cluster with only a single compute node as Neo4j needs at least two nodes in a cluster. Because of the same reason, we compare the query latency results of Neo4j and GDI using 2 compute nodes. Benchmark node configurations and the core/replica node ratios are the same as the previous experiments.

Figure 7 shows us quite a different picture than the latency histograms of JanusGraph. Here, we can see that instead of overlapping completely, the latency histograms of queries appear to be shifted towards right as the number of compute nodes and queries increase. Not only that but the tail latencies of queries are significantly higher than that of JanusGraph or GDI. The update vertex subplot shows us that the tail latency is around 25000ms. This number is upwards of 40000ms for adding edges to the graph. These results further emphasize that Neo4j struggles to scale.

Figure 8 tells us a similar story as the weak and strong scaling experiments of Neo4j. Each query executed on Neo4j is around 3 orders of magnitude slower than GDI.

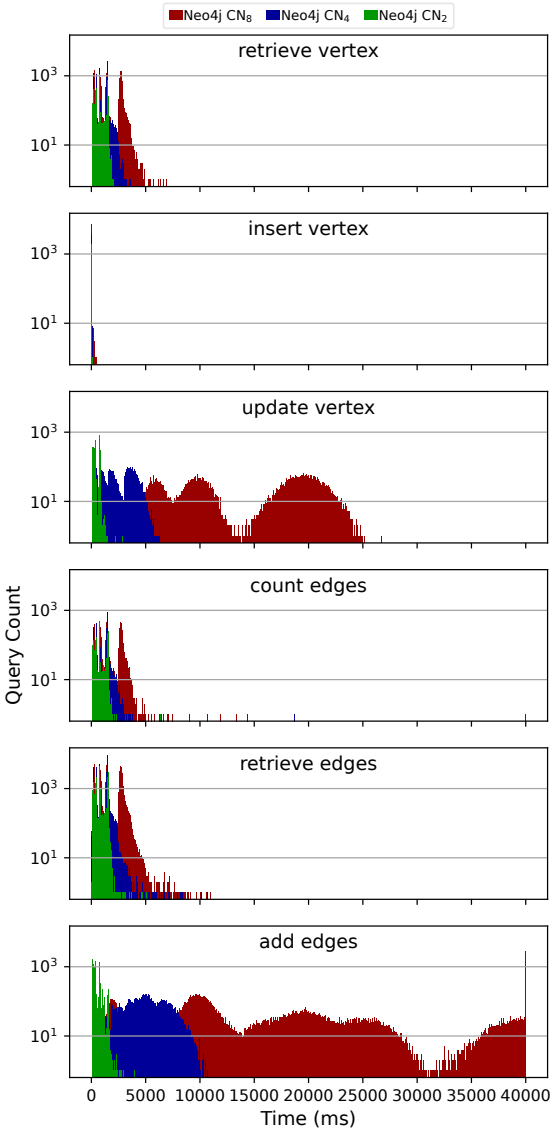

Fig. 7: *Latency*: Comparison of Neo4j query latencies with different number of cluster nodes.

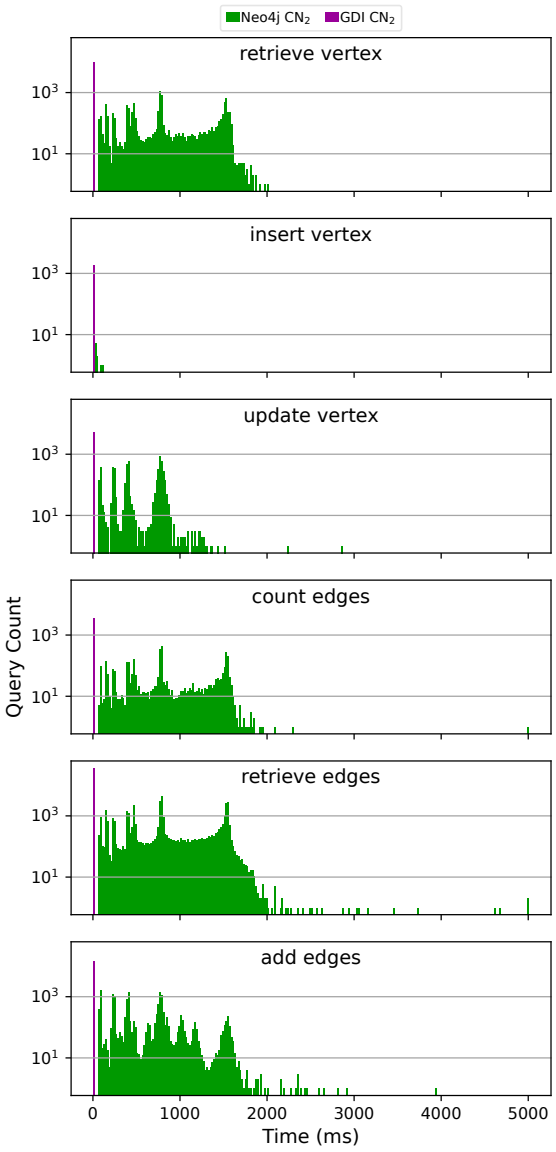

Fig. 8: *Latency*: Comparison of Neo4j and GDI query latencies with a single node.

5. CONCLUSIONS

We have presented benchmarking results of JanusGraph & Neo4j and compared them to the performance of GDI. JanusGraph linearly scales with the number of nodes in the cluster up to half a million queries per second while the best Neo4j throughput we achieved was up to 100 queries per second. Neo4j was unable to scale horizontally and its performance degraded as the number of machines in the cluster and the queries issued increased. GDI has an order of magnitude higher throughput than JanusGraph and it has three orders of magnitude higher throughput than Neo4j. Both JanusGraph and GDI suffer from increased rates of query failures as the number of queries in the system increase unless the graph sizes scale proportionally to the number of queries.

6. FUTURE WORK

In order to better understand the performance of the benchmarked systems and use them as a baseline against GDI, additional query performances can be measured. An important class of algorithms used to measure the performance of graph database management systems as well as high performance computing systems are graph traversal algorithms, namely k-hop and breath first search algorithms. We can extend the benchmarks to measure the run times of the systems executing these algorithms. Additionally, other open source graph databases like TigerGraph can be benchmarked to get a more comprehensive picture of the performance GDI against current GDBs.

7. REFERENCES

[1] Brugnara M. Velegrakis Y. Lissandrini, M., "Beyond macrobenchmarks: Microbenchmark-based graph database evaluation," *Proc. VLDB Endow.*, vol. 12, no. 4, pp. 390–403, dec 2018.

[2] Urbón-Bayes P. Giménez-Vañó A. Gómez-Villamor S. Martínez-Bazán N. Larriba-Pey J.L. Dominguez-Sal, D., "Survey of graph database performance on the hpc scalable graph analysis benchmark," in *Web-Age Information Management*. 2010, vol. 6185 of *Lecture Notes in Computer Science*, p. 37–48, Springer, Berlin, Heidelberg.

[3] Bernardino J. Fernandes, D., "Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb," in *Proceedings of the 7th International Conference on Data Science, Technology and Applications*, Setubal, PRT, 2018, DATA 2018, p. 373–380, SCITEPRESS - Science and Technology Publications, Lda.

[4] Sá F.-Bernardino J. Monteiro, J., "Graph databases assessment: Janusgraph, neo4j, and tigergraph," in *Perspectives and Trends in Education and Technology*. jan 2023, vol. 320 of *Smart Innovation, Systems and Technologies*, Springer, Singapore.

[5] Guo X.-McMillian B. Qian-K. Wei MY.-Xu Q Wasserman, A.I., "Osspal: Finding and evaluating open source software," in *Open Source Systems:*

Towards Robust Practices, Di Cosmo R. Garrido-A. Kon F.-Robles G. Zacchiroli-S. (eds) In: Balaguer, F., Ed. IFIP Advances in Information and Communication Technology, Springer, Cham.

[6] Feo J. Gilbert J. Kepner-J. Koester D.-Loh E. Madduri-K. Mann B.-Meuse T. Bader, D.A. and E. Robinson, "Hpc scalable graph analysis benchmark," *Citeseer*, pp. 1–10, 2009.

[7] "Janusgraph," https://docs.janusgraph.org/.

[8] Apache Cassandra, "Apache cassandra homepage," https://cassandra.apache.org/.

[9] Ponnekanti V. Borthakur D. Armstrong, T. G. 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*, New York, NY, USA, 2013, SIGMOD '13, p. 1185–1196, Association for Computing Machinery.

[10] "Graph500," https://graph500.org/?page$_id = 12$.