Comparative Analysis of Graph Databases for Git Data

Thiago A. Falcâo*1a, Phillip M. Furtado1b, Jordan S. Queiroz1c, Paulo J. O. Matos1d, Tiago F. Antunes1e, Frederico S. Carvalho1f, Paulo C. R. Fonseca1g, and Felipe T. Giuntini1h
1Sidia R&D Institute, Manaus, AM, Brazil

Corresponding author: *thiago.falcao@sidia.com; bphillip.furtado@sidia.com; cjordan.queiroz@sidia.com; etiago.antunes@sidia.com; dpaulo.oliveira@sidia.com; ffredierico.carvalho@sidia.com; gpaulo.fonseca@sidia.com; hfelipe.giuntini@sidia.com

Abstract. In past decades, the requirements that database management systems (DBMSs) must achieve have become increasingly stringent (speed, data volume). This increase in complexity led to the development of a wide range of non-relational databases strategies, each one suited for specific scenarios. In this context, Graph Database Management Systems (GDBMSs) became popular to represent social networks and other domains that can be intuitively represented as graph-like structures. In this paper, we represent Version Control System data, specifically Git, from a large software project in a graph structure and compared three popular GDBMSs: Neo4j, JanusGraph and Dgraph. We evaluated read/write operations performance for common activities, such as inserting new commits into the graph and retrieving the complete commit history of a specific project. With this contribution, researches and engineers may choose, assertively, the better solution for their needs.

1. Introduction
Current performance requirements regarding data access have contributed to the development of new types of database management systems (DBMSs). Many of these DBMSs paradigms are non-relational and specific-purpose, aiming to leverage their performance by exploiting characteristics of a particular area, such as time series [1], documents [2] and graphs [3].

Since the continued emergence and increase of massive and complex graph-like data makes a graph database a crucial requirement [3], several solutions have been developed. These systems are called GDBMSs (Graph Database Management Systems).

A graph database is a type of NoSQL database based on the graph abstraction to naturally represent data as nodes and edges. Application domains such as social networks, Semantic Web, geographic applications and bio informatics are examples which data have natural representations in terms of graphs. [4]

Git is a version control system (VCS), which is responsible for managing changes to computer programs. The data derived from the development of a project using Git can be represented as a DAG (Directed Acyclic Graph). Besides, Git data has strongly related elements and for this scenario, according to the literature [5], graph database has superior performance.
In this paper we compared three different GDBMSs on the domain of Git data: Neo4j, JanusGraph and Dgraph. The systems to be compared were chosen considering their relevance in the community and industry. We evaluated the performance of write and read operations for common activities, such as inserting new commits into the graph and retrieving the complete commit history of a specific project. With this contribution, researchers and engineers may choose, assertively, the better solution for their needs.

2. Related work
A number of efforts in the literature also analyzed features and characteristics of different GDBMSs in general and particular scenarios. Some works compared graph databases and other relational and non-relational databases. Almabdy [5] compared relational and graph databases for social networks. The comparison considers storage, query language, query performance and transaction. To achieve this, the author uses a Twitter dataset and Neo4j and MySQL databases. Das et al. [6] reviewed ten databases and compared them regarding the following features: multi-architecture, query language, multi-model, flexible schema, sharding, scalability, backup, portability and ACID support. To evaluate the features, the authors use Likert scale.

Other authors focused on comparing only graph databases. Angles [3] compared nine graph databases according to their storage (memory, backend storage and indexes), operation and manipulation features (query language, API, GUI), data structure (graphs, nodes and edges) and other metrics. Ciglan [7] addressed the need to benchmark the performance of five different graph databases comparing the ability to efficiently traverse the graph topology using traversal operations and concluded that traversal of the whole graph are feasible when the graph is cached in memory. McColl [8] compared the performance and qualitative features of 12 open source graph databases using four graph algorithms based in a network containing up to 256 million edges.

Other studies analyzed the performance of a graph database in a specific scenario. Geiger et al. [9] created a dataset of Android apps with metadata from GitHub projects and repositories, and from Google Play store. To store the data, the authors used Neo4j. The data is modeled as a graph and the dataset contains data about 8,431 Android apps.

Similarly to the above related works, this paper compares graph databases using as data source git data. However, instead of supported features, this paper compares performance, such as the executing time for read and write operations. Like [5], this paper compares databases according to theirs performance, but in this paper, all of them are graph databases.

3. Background
This section briefly describes the main concepts of Git and graph databases.

3.1. Git
A version control system (VCS) is responsible for managing changes to computer programs by recording changes to a file or set of files over the time so it is possible to access specific versions later. Git is a particular type of VCS called distributed version control system (DVCS). This class of VCS allows multiple users to store their own version history without the need to synchronize with a centralized store. The codebase and its history are mirrored on every users' computer. Instead of checking out the latest snapshot of the files, each local version fully mirror the repository, including its full history.

Git has some key concepts that leverages software development, to name a few:

- Repository: Contains a collection of files of various different versions of a Project. Each project has its own repository.

https://www.geeksforgeeks.org/what-is-a-git-repository/
- **Commit**: Represents a change in the repository in a specific time\(^3\). Any changes must be turned into a commit in order to be safely saved in the repository.
- **Branch**: Every project has a main line, also known as branch. Branch allows working separately in new software features without mutating the main line. Once the new features are done, the secondary branch can be merged into the main one.
- **Tag**: It is generally used to capture a point in history of the project, for example, a new release. Unlike branch, after being created, it have no further history of commits\(^4\).
- **User**: It is a collaborator in the project. Users can create tags, branches, commits, new repositories and so forth.

The figure 1 illustrates an overview of git repository with a typical workflow.

**Figure 1.** Git Graph example

The rectangles **Main**, **Develop** and **Features** are branches. The **Main** is the primary one. Each branch points to a commit, represented by the circles. Blue circles are commits from branch **Main**, purple circles are commit from branch **Develop** and so on.

As the software evolves (from left to right), new commits are made in different branches. This way the code splits (first green circle) and new features are added. Later it merges (second green circle) with the ongoing work in another branch.

Tags (red circles) may be used like a check point for versions, like v1, v2, alpha, beta and so on. All this events happen in a repository, where a project is developed by users.

### 3.2. Graph Database

Graph Databases store data as a graph, which is a set of vertices and edges. In graph databases there are two main concepts\(^4\):

- **Nodes (vertices)**: They are the entity in the graph, the data itself. They can have any number of attributes, also called properties, stored as key-value pairs. A node can also have a label, representing its role in the database.

---

\(^3\) [https://www.atlassian.com/git](https://www.atlassian.com/git)

\(^4\) [https://neo4j.com/developer/graph-database/](https://neo4j.com/developer/graph-database/)
• Relationships (edges): Establish a relationship between two or more nodes in the graph. A relationship always has a direction, a type, a start and end node. Similarly to nodes, relationships can also have properties. In a graph database, a node can have as many relationships as necessary.

The figure 2 illustrates a basic graph model in graph database. In the figure, the nodes Employee, Company and City represents the data. The edges HAS_CEO and LOCATED_IN represent relationships between data. Both nodes and edges may have properties and labels.

![Figure 2](image2.png)

**Figure 2.** Simple graph model, representing the relationship of a company between an employee and a city.6

### 4. Git and Graph Features Comparison

As we can see in figure 1, git structure is naturally represented by a graph, so it can easily be stored on a graph database.

We make Git data easily available and searchable by creating and populating a graph according to the figure 3 that illustrates our Git graph model used in experiments.

![Figure 3](image3.png)

**Figure 3.** Our Git graph model

Table 1 describe the mapping of Git characteristics to a GDBMS:

For example, PARENT_OF edge will become very deep by the fact that a repository can have many commits and each commit has a parent, except the first one. In the next section, we evaluate how the particularities of a graph based on Git data affect the performance when dealing with these types of scenarios.

### 5. Experimental Analysis

This section presents the steps that we used to execute our experiment and extract the results.
5.1. Graph databases
As mentioned before, we have conducted experiments with three Graph Databases: DGraph, JanusGraph and Neo4J. These databases were chosen considering their relevance in the community and industry; and they only work with data modeled as graphs.

Dgraph is a distributed, low-latency, high throughput graph database with emphasis on design and concurrency. It is consistent and fault tolerant. Dgraph provides high availability through data sharding and horizontal scalability. The query language used in Dgraph is GraphQL+, a variation of GraphQL. It has a distributed architecture and it is divided into three parts: Dgraph Zero, which controls the cluster; Dgraph Alpha which hosts predicates (data) and indexes; and Dgraph Ratel, which serves the UI to run queries.

Table 1. Nodes and edges of the graph.

| Name            | Type  | Description                                      |
|-----------------|-------|--------------------------------------------------|
| COMMIT          | Node  | A git Commit                                     |
| USER            | Node  | An user                                          |
| TAG             | Node  | A git tag                                        |
| BRANCH          | Node  | A git branch                                     |
| HAS_BRANCH      | Edge  | A commit can have a branch                       |
| HAS_TAG         | Edge  | A commit can have a tag                          |
| TAGGED_BY       | Edge  | An user can create many tags                     |
| PARENT_OF       | Edge  | A commit can be parent of one or more commits    |
| AUTHORED_BY     | Edge  | An user authors many commits                     |
| COMMITTED_BY    | Edge  | An user commits many commits                     |

JanusGraph is a scalable graph database optimized for storing and querying large graphs distributed across a multi-machine cluster. It can scale with the number of machines in the cluster, supporting many concurrent transactions and operational graph processing. The query language used in JanusGraph is Gremlin. It provides data distribution and replication for performance and fault tolerance. Also supports many data storage, for example Apache Cassandra, Apache HBase and ScyllaDB. Advanced search capabilities can optionally be supported by Elasticsearch and Apache Lucene. In this paper, we used Apache Cassandra as data storage.

Neo4j is a graph database powered by a native graph storage and processing engine. It provides High Availability with Raft-base casual clustering rolling upgrades and hot backups. Neo4j scales in two ways: (I) sharding, which distribute large datasets into smaller physical database and (II) query federation, which enables queries across disjointed graphs. It also provides Fine-grained security, LDAP/Directory services, security logging. ACID transactions is supported by Neo4j, providing reliability. The query language used is Cypher, which is used in the UI that comes with Neo4j. Regarding memory usage, official documentation recommends to set heap size to at least 8GB to run Neo4j reliably.

With exception of Neo4j, all databases are fully distributed. Fault tolerance and high availability are supported in all databases, which can be run in many operating systems. Differences among the databases are found in RAM consumption and in query language.

Table 2 summarizes the key points of these three databases.

---

7 https://dgraph.io/docs
8 https://janusgraph.org/
9 https://neo4j.com/neo4j-graph-database/
10 https://neo4j.com/developer/guide-performance-tuning/
Table 2. Overview of Graph databases.

|                  | Neo4j                  | JanusGraph            | Dgraph             |
|------------------|------------------------|-----------------------|--------------------|
| Distributed      | Not fully distributed  | Yes                   | Yes                |
| Fault Tolerant   | Yes                    | Yes                   | Yes                |
| High Availability| Yes, with sharding     | Yes, with replication | Yes, with sharding |
| Query Language   | Cypher                 | Gremlin               | GraphQL+            |
| Runs on          | Linux, Windows, Mac, Container | Linux, Windows, Mac, Container | Linux, Windows, Mac, Container |
| RAM consumption  | Up to 50% of RAM for caching | Depends on the JVM    | Up to 80% of RAM   |

5.2. Dataset
It was used as data source a Gerrit server version 2.14 with 819 git repositories and 14126 users. The size of all this data is around 50GB.

5.3. Methodology
The environment was one machine with Server Operating System Ubuntu 16.04, 396GB of RAM, 64 CPUs and 10TB of available hard disk storage, where it was installed the graph databases.

DGraph version v20.07.1 was installed, which it is a cluster of 3 different nodes: Zero, Alpha e Ratel nodes.

For JanusGraph was installed JanusGraph 0.5.2 as graph engine and Apache Cassandra 3.11.8 11 as data storage. Apache Cassandra is a distributed, wide column store, NoSQL database management system where our graph data is stored.

In the case of Neo4J, it was installed only Neo4J 4.1.3, which includes user interface and server.

After installation, it was drawn the graph model as already described in figure 3.

We developed a script in the Python12 programming language in order to extract data through git commands and store it in target databases as vertices and edges. In this step, we collect writing performance results.

First the data is collected from a Git Server. All the data are structured as lists, then the collector send those data to the handlers of each graph database: Neo4JProxy, JanusProxy and DgraphProxy.

The handlers are responsible for transforming the collected data into vertices and edges. Each handler has specific methods to create users, commits, tags and branches vertices. Besides vertices, each handler also make the relations between vertices through edges. Similarly to vertices, there are specific methods to create the edges has_branch, has_tag, tagged_by, authored_by, committed_by and parent_of.

After the collected data is processed into vertices and edges, the handlers save those data into the database. Each handler is specific to each database, for example, JanusProxy only sends data do JanusGraph. The figure 4 summarizes how the collector script with handlers works. With the database populated, the next step was to perform query operations.

5.4. Results and discussion
We conducted ten experiments for write operations: create vertices (commit, user, tag and branch vertices) and create edges (has_branch, has_tag, tagged_by, authored_by, committed_by and parent_of).

Five experiments were performed for read operations: get all commits from a specific repository, get all branches from a specific repository, get all tags from a specific repository, get all commit children given a specific commit hash, and finally, get all commit authors between two different tags.

11 https://cassandra.apache.org
12 https://www.python.org
Figure 4. Collector script workflow.

Performance results are extracted using Python Profiling\textsuperscript{13}. A profile is a set of statistics that describes how often and for how long various parts of the program executed. So, we used it to measure the time taken for write and read operations of nodes and edges.

Figures 5 and 6 show the performance results on creating vertices and edges, respectively. We can see that Neo4J outperforms all graph databases on write vertices and edges. Figure 7 shows the performance results on read operations. We can see that Neo4J outperforms all graph database, except on getting tags.

It was used indexing feature of all graph database in order to speed up read operations. But, we could conclude that Neo4J indexing is better, because it has composite indexing. Composite indexes are very fast and efficient but limited to equality lookups for a particular, previously-defined combination of property keys. JanusGraph and Neo4J used composite indexes combining: Vertex user and username field; commit vertex, repository name and commit hash fields; tag vertex, repository name and ref name fields; and branch vertex, repository name and ref name fields. DGraph also uses indexing to speed up queries, but does not have composite indexing.

\textsuperscript{13}https://docs.python.org/3/library/profile.html
which means it is only possible to index each field individually. Neo4J and Dgraph were much better than JanusGraph by the fact they have their own internal database, while JanusGraph used an external data storage (Apache Cassandra).

**Figure 5.** Performance on creating vertices in ms.

**Figure 6.** Performance on creating edges in ms.

6. Conclusion
We made a performance evaluation of three modern graph databases using git history as data source. This study served as a guidance on how to model graphs, nodes and edges, and compare graph databases. The graph representation of a git project makes easy the real time analysis of commit networking, commit parenting, branches and tags activities and user contributions.

After the comparison, we end up in a conclusion that Neo4j outperforms the other graph databases like JanusGraph and Dgraph, providing a much faster read and write performance, making it a suitable option for very high load applications and at the same time been a popular graph database with an easy graph language, Cypher, and the Neo4j Desktop tool. On the other
Figure 7. Performance on read operations in ms.

hand, JanusGraph had the worst results in this comparison. Dgraph, when compared against JanusGraph, is close to Neo4j.

7. Acknowledgements
Although he was not directly involved in this work, we would like to thank Mr. Byoung-Ho Gu for his contributions that made it possible. This work was partially supported by Samsung Eletrônica da Amazonia Ltda, under the auspice of the informatic law no 8.387/91. This work was supported by the Sidia: Institute of Science and Technology.

8. References
[1] Bader A, Kopp O and Falkenthal M 2017 Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband
[2] Han J, Haihong E, Le G and Du J 2011 Survey on NoSQL database 2011 6th international conference on pervasive computing and applications (IEEE) pp 363–366
[3] Angles R 2012 Proceedings - 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012 171–177
[4] Virgilio R, Maccioni A and Torlone R 2014 Model-driven design of graph databases International Conference on Conceptual Modeling (Springer) pp 172–185
[5] Almabdy S 2018 Comparative analysis of relational and graph databases for social networks 2018 1st International Conference on Computer Applications Information Security (ICCAIS) pp 1–4
[6] Das A, Mitra A, Bhagat S N and Paul S 2020 Issues and concepts of graph database and a comparative analysis on list of graph database tools 2020 International Conference on Computer Communication and Informatics (ICCCI) pp 1–6
[7] Ciglan M, Averbuch A and Hluchy L 2012 Benchmarking traversal operations over graph databases 2012 IEEE 28th International Conference on Data Engineering Workshops (IEEE) pp 186–189
[8] McColl R C, Ediger D, Poovey J, Campbell D and Bader D A 2014 A performance evaluation of open source graph databases Proceedings of the first workshop on Parallel programming for analytics applications pp 11–18
[9] Geiger F, Malavolta I, Pascarella L, Palomba F, Di Nucci D and Bacchelli A 2018 A graph-based dataset of commit history of real-world android apps 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR) pp 30–33