Available online at sjuoz.uoz.edu.krd

Science Journal of University of Zakho

Vol. 11, No. 2, pp. 190 –201, April-June, 2023

p-ISSN: 2410-7549
e-ISSN: 2414-6943

AN EMPIRICAL COMPARISON OF NEO4J AND TIGERGRAPH DATABASES FOR NETWORK CENTRALITY

Bahzad Taha Chicho a*, Abdulhakeem Othman Mohammed b
a Duhok Polytechnic University, Technical College of Informatics, Information Technology Department, Kurdistan Region, Iraq - bahzad.taha@dpu.edu.krd
b Duhok Polytechnic University, Kurdistan Region, Iraq - abdulhakeem.mohammed@dpu.edu.krd

Received: 19 Nov., 2023 / Accepted: 30 Apr., 2023 / Published: 30 Apr., 2023 *https://doi.org/10.25271/sjuoz.2023.11.2.1068*

ABSTRACT

Graph databases have recently gained a lot of attention in areas where the relationships between data and the data itself are equally important, like the semantic web, social networks, and biological networks. A graph database is simply a database designed to store, query, and modify graphs. Recently, several graph database models have been developed. The goal of this research is to evaluate the performance of the two most popular graph databases, Neo4j and TigerGraph, for network centrality metrics including degree centrality, betweenness centrality, closeness centrality, eigenvector centrality, and PageRank. We applied those metrics to a set of real-world networks in both graph databases to see their performance. Experimental results show Neo4j outperforms TigerGraph for computing the centrality metrics used in this study, but TigerGraph performs better during the data loading phase.

KEYWORDS: Graph Database, Relational Database, Database Model, Neo4j, TigerGraph.

1. INTRODUCTION

Nowadays, the development, consumption, and, most importantly, the analysis of highly correlated data have become pervasive. On a daily basis, the volume of data continues to expand rapidly on social networks such as Facebook, Twitter, etc., which store and analyse huge amounts of data, approaching petabytes of storage. When the quantity and relevance of data links increase concurrently, graph models seem like a smart notion [1]. Since most real-world data can be simply represented on graphs, there has been a growing interest in graph representation in recent years [2]. At the same time, a graphical representation of data is an appealing method of displaying numerical data that aids in quantitative data analysis and visual representation.

Graphs are a good way to show both structured and unstructured data, and they can be thought of as a unified way to show data. Multiple domains, including the Semantic Web [3], images, social networks [4], bioinformatics, etc., can be naturally modelled as graphs. So, recent research on databases shows that there is a growing interest in building graph models and languages to help these applications manage information [5]. In fact, a graph database facilitates more natural modelling and provides flexible support for dynamic data.

Graph database is optimal for addressing complex, semi-structured, and heavily interconnected data. It provides a response in milliseconds, and queries are processed incredibly quickly [6]. Graph databases are very useful at enterprise levels, such as in communication, healthcare, retail, finance, social networking, online business solutions, online media, and so on. Moreover, a graph database, also known as a "semantic database," is a software application that stores, queries, and modifies network graphs. The components of a network graph are nodes and edges. Each node indicates an entity (like a person), whereas each edge indicates a relationship between two nodes [7]. There are many graphical database tools, each with its own set of capabilities and performance.

As for the relation of the Graph Database (GD), known as the NoSQL database, to the traditional database management systems (DBMS), it is completely different as it depends on relationships rather than foreign keys as its foundation. It enables dealing with more expressive data, such as phylogenetic tree topologies, and explicitly supports querying on a data network. These databases are very flexible with real-world data that is continually changing. The capacity of graph databases to execute reasonably consistent large-scale join-query operations as the dataset expands is another important benefit they have over relational databases [8].

GD models are utilised in domains where understanding the topology or interconnection of the data is as crucial as the data itself. The data and its connections in these applications are often on the same level [9]. In this study, we focused on both Neo4j and TigerGraph, the two most widely used databases in the field of graphs, and the following are the study's contributions:

1. Demonstrates the capabilities of the Neo4j and TigerGraph databases.
2. Compares the response time of Neo4j with TigerGraph based on the used metrics.
3. Shows the response time of the data loading phase for both Neo4j and TigerGraph.

The remaining sections of the paper are structured as follows: Related Works are introduced in section 2; Graph Databases are illustrated in details in section 3; Graph Databases tools are explained in section 4; Network centrality and used metrics have been clarified in section 5; Dataset Description is described in section 6; Experimental Results are discussed in section 7. Finally, we conclude the paper and discuss future research directions in section 8.

* Corresponding author

2. RELATED WORKS

In this section, various recent studies on graph databases are reviewed.

Lu et al. [10] analysed film data using Neo4j and Cypher Language. The essential method for analysing film data is database traversal and querying. Neo4j prioritises performance in standard Relational Database Management System (RDBMS) operations with many connections. Then, the Neo4j is used to examine relationships between key items in the film data, such as directors, actors, and so on. The experimental results show that Neo4j is capable of handling complicated data that has many connections.

Almabdy [11] provided a comparative investigation of Neo4j and the Relational Database Management System using a Twitter dataset, comparing their overall performance and capability. The research group specifically proposed an experiment to evaluate the performance, operating scope, and overall usefulness of the two graph databases. While the RDBMS searches through all the data to satisfy this requirement that matches the search criteria. By contrast, the Neo4j search returns only records that are inextricably linked to other data. As a result, the relational database took longer to create. Neo4j was shown to be quicker and more flexible than RDBMS.

Lysenko et al. [12] explained how GD prepares a strong foundation for storing, retrieving, and visualising biological data. Where human protein-protein interactions, pathways, sequence similarity, disease-gene, gene-tissue, and protein-drug connections were all used in the investigations. The system built and queried a prototype illness map for a complicated ailment like asthma using the well-known Neo4j graph database. The results revealed that GDs are appropriate for storing biological data, which is often densely linked, semi-structured, and unpredictable.

Šestak et al. [13] proposed a new idea for k-vertex cardinality constraints, which allows for specifying the minimum and a maximum number of edges between a vertex and a subgraph. The study suggests an approach that involves encoding cardinality restrictions to overcome the challenge of expressing cardinality constraints in GDs. The approach uses a property graph data model and demonstrates its execution in Neo4j Graph Database Management Systems through a series of stored procedures (GDBMSs). The suggested technique is then tested on synthetic and real datasets to determine how verifying cardinality restrictions affect Query Execution Times (QETs) when adding new edges. A comparison of synthetic datasets with different outgoing vertex degrees was carried out. The findings show that the outgoing vertex degree of the datasets and the order of the underlying k-vertex cardinality restrictions affect the model.

Macak et al. [14] contrasted the OrientDB multi-model database with the Neo4j GD and MongoDB document database. Several queries were used to determine the performance of the used databases. These queries are separated into two distinct groups: the first one is associated with the graph data, while the second one is associated with the document data. Neo4j was used to compare with OrientDB to determine the shortest path traversal between nodes with different depths for a graph query. In terms of document queries, they compared OrientDB to MongoDB to highlight the distinction between querying on an indexed and non-indexed field. The findings indicated that Neo4j was the best option for traversing different nodes up to three depths through graph queries. Aside from that, OrientDB was the best option. For document queries, MongoDB outperforms OrientDB in document data management.

Mitali Desai et al. [15] presented an extensive empirical analysis of Neo4j, ArangoDB, and OrientDB graph databases. In order to identify influential entities from Twitter data, query response time is used to evaluate the efficiency and effectiveness of primitive indexing approaches. The results demonstrate that Neo4j executes loading, relationship, and property queries more efficiently and reliably than the other two databases. While primitive indexing could be employed to increase OrientDB's efficiency.

Beis et al. [16] presented a benchmark to evaluate Titan, OrientDB, and Neo4j GDs. In the study, real and synthetic networks were both employed, and the execution time was used to compare the two. In which the study concentrates on the issue of community detection and employs the Louvain method. The findings indicate that OrientDB is the quickest option for the Louvain method, whereas Neo4j is the fastest for query workloads. Also, Neo4j and Titan are better at handling large numbers of insertions and single insertions, respectively.

R. Wang et al. [17] provided a thorough analysis and empirical study of the property GD systems, including Neo4j, AgensGraph, TigerGraph, and LightGraph. The research was done in a single-machine setting using the Linked Data Benchmark Council Social Network Benchmark (LDBC SNB). The LDBC SNB consists of three different large-scale datasets (DG1, DG10, and DG100) as well as a set of queries to evaluate performance. Based on the queries used, they systematically evaluated all the used graph databases in terms of data loading and query performance according to the used datasets. The results show that AgensGraph performs well for SQL-based workloads and simple update queries, and TigerGraph works well for complex business intelligence queries. While the Neo4j is easy to use and works well for small queries, the LightGraph is a more versatile product for many queries.

C. Liu and H. Duan [18] presented a high-performance graph database storage engine for provenance graphs called Temporal Dimension-Graph Database (T-GDB). The system connects the graph's topology to each vertex and reconstructs the graph in real-time. The T-GDB may also access the provenance of the given graph via the index tree and evaluate how a graph evolves over time. To confirm the viability and effectiveness of the approach, TigerGraph, Neo4j, and JanusGraph were used to compare with T-GDB on the Graph500 and com-Orkut datasets. The results of the evaluation showed that the T-GDB storage engine does graph analysis better than other methods.

T. Chen et al. [19] suggested a unique graph computing framework-based simulation platform on TigerGraph to ease the energy market clearing procedure. The platform was implemented using graph parallelised power flow for Security Constrained Unit Commitment (SCUC) and Security Constrained Economic Dispatch (SCED) issues. TigerGraph's GSE/GPE blocks provide several adaptable interfaces for storing and managing market or network data. In addition, a sophisticated visualization platform based on GraphStudio is utilised to illustrate the outcomes of the electrical market clearing and the transmission congestion impact. The results show that the system as a whole can be used to educate and train energy market operators.

F. Rusu and Z. Huang [20] the two native GD tools, Neo4j and TigerGraph, were used to evaluate the bulk loading time and store size, and the outcomes of an application guideline for the LDBC SNB were reported. In addition, it evaluates the performance of all queries used in the benchmark on four different sizes of scale factors that were utilised as datasets on-premises and in the cloud. TigerGraph routinely outperforms Neo4j on the great majority of queries, exceeding 95% of the workload, according to the data. In a fair amount of time, Neo4j completes just 12 of the 25 business intelligence queries. Still, the difference between the two systems widens as the quantity of data rises, as only

TigerGraph can expand to SF-1000. But Neo4j is faster at loading large amounts of graph data (not counting the time it takes to build the index) and has a more concise declarative query language.

A. Deutsch et al. [21] revealed how Graph Structured Query Language (GSQL), the graph query language of TigerGraph, facilitates the definition of aggregation in graph analytics. The used datasets correspond to the LDBC SNB standard, with graph sizes ranging from 1GB to 1TB and scale factors ranging from 1 to 100 on an Amazon EC2 instance of type r4.8xlarge. Whereas the GSQL has a number of distinctive design considerations regarding the expressive capacity and evaluation difficulty of the defined aggregate. The results show that TigerGraph's GSQL supports aggregate requirements in graph analytics and is easy to convert to upcoming graph query language standards and pattern-based declarative query languages.

3. GRAPH DATABASES

In today's world, technology is advancing at a breakneck pace, and we are enabling the advantages of linked data. A GD is the best way to deal with data that is well-organised, semi-structured, and heavily linked. It gives responses in milliseconds and executes queries very quickly [22]. At the corporate level, GDs are very valuable in fields such as communication, healthcare, retail, financial, social networking, online business solutions, online media, etc. A GD system implements the CRUD (Create, Read, Update, and Delete) operations found in a graph data model, as well as index-free adjacency, as shown in Figure 1.

Adjacency without indexes is critical for high-performance traversal. If a graph database makes use of this, each node keeps a direct reference to its neighbours. It is referred to as a "micro index" for other nodes and is far less expensive than global indexes. This implies that query time is independent of the graph's overall size and is purely relevant to the duration of the graph search. This implies that the database's linked nodes constantly point to one another. It is very quick in terms of query response time and also has the capacity to hold massive volumes of data.

GDs do not use tables to store data. It has a single data structure – the graph – and there is no join procedure, which means that each vertex or edge is directly linked to another vertex. A graph organises data into nodes that are connected by a few relationships. Graph databases are organised according to the property graph model and also developed for use in transnational Online Transaction Processing (OLTP) systems. These are intended to ensure the integrity of transactions and operational availability. At the moment, known graph databases are classified as NoSQL databases. A performant graph database paradigm is required for improved graph management.

The property graph paradigm presents an alternative approach to data model representation in graph theory. This paradigm has introduced a standard for visualising property graph-based data models, known as Graph Data Modelling. Property graphs are defined as graph data structures that comprise nodes and relationships, with attributes that may exist in either the nodes or relationships. This feature offers greater flexibility in attribute placement, enabling more nuanced graph-based data modeling. Therefore, the property graph approach facilitates a paradigm shift in graph theory and enhances the visualisation of graph-based data models. Such models are more directly tied to the user's issues in graph databases. These approaches are straightforward but costlier than relational databases and other NoSQL databases. Due to their capacity to handle graph-like structures in modern applications, graph databases have regained prominence in the modern period, earning them the moniker "the future of database management systems.

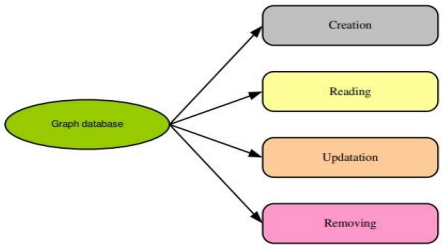

Figure 1: Units of Graph Database [23].

3.1 Query Language

In general, a query accepts a graph as input (referred to as the target graph), matches graph patterns, and generates a table of results. Three sample query languages were combined to create the language's syntax (i.e., SQL, SPARQL, and Cypher). A non-recursive, safe datalog with negation transformation is used to specify the language's semantics [24].

Syntax

The query language's syntax is organized around four fundamental clauses: SELECT, FROM, MATCH, and WHERE. These clauses enable you to write queries, including simple pattern matching. Furthermore, the query language has the UNMATCH and UNION phrases to facilitate the negation and union of graph structures, respectively.

The language's primary characteristic is pattern matching, which allows you to define graph patterns that are matched against the target graph. The input graph is described in the FROM clause, a graph pattern is specified in the MATCH clause, and conditions are applied to the input graph in the WHERE clause. The SELECT clause specifies the query's output (in this case, a table of data). An example of a pattern matching query is shown below. The result of the search is a list of authors who have written articles together. The results of the query are the authors' names who have written articles together.

1- *SELECT* n2.fname AS Author1, n3.fname AS Author2, n1.title AS EntryTitle.
2- *FROM* "biblio".
3- *MATCH* (n1:Entry) - [e1:has - author] → (n2:Author), (n1) - [e2:has - author] → (n3).
4- *WHERE* n2 != n3 AND e1.order < e2.order.

3.2 Advantages and Disadvantages of Graph Database

Graph databases have become increasingly popular in recent years due to their ability to handle complex and interconnected data efficiently. However, like any other technology, they have their advantages and limitations that must be considered before implementation.

Advantages: Graph databases (GDBs) offer several advantages, including object-oriented thinking, improved problem-solving capabilities, a flexible online schema environment, robust AI infrastructure, efficient indexing, and scalability. GDBs allow for explicit and clear semantics for each query, making it easier to understand and manipulate data. They are capable of solving both practical and theoretical problems, including those posed by iterative and ML algorithms. The dynamic online schema environment provides the ability to add or remove additional vertex or edge types and characteristics, making it easy to expand or constrict the data model. The indexing of GDBs is logical and based on relationships, which enables faster access than relational databases. Additionally, GDBs are scalable and can handle large datasets and enhance both reading and writing

performance. Therefore, GDBs are a powerful tool for data scientists and other professionals who work with complex data sets.

Neo4j, a popular graph database management system, has implemented two levels of caching to address the issue of scalability when handling highly concurrent workloads. Vertical scaling is an option to increase the size of the writing, but it may face challenges when dealing with very dense writing masses. In such cases, it becomes necessary to distribute the data across multiple computers to achieve scalability, which poses significant challenges.

Disadvantages: Like any other database, GDs face issues because of the wide availability of large amounts of data. In other words, fast-growing, fast-paced, and poorly organised volumes of data require high-performance IT solutions to effectively analyse and use them. GDB does not combine strong competitiveness with serviceability. Enhancements to associations are not generated by graph databases; they just serve as a quick repository for linked information. While promoted search strategies are many, they would not be considered in the case of connections since they would not effectively detain anyone in the first place. Given the properties of data storage, GDs are not ideal for large-quantity analytic queries. GDs are less useful for operational use cases because they perform poorly when managing large numbers of transactions and do not perform better when performing queries that comprise the entire database [25].

3.3 Why a Graph Database Model?

GD models are used when the topology or connection of the data is equally or more relevant than the data itself. In GDB operations, the relationships between the data and the data themselves are often at the same level. In fact, adding graphs as a way to model the graph database has a number of benefits [26].

1. It results in more realistic modelling. Graph topologies prepare a natural way to manage application data that is available to users (e.g., hypertext or geographic databases). Using connected arcs to represent related information, graphs offer the advantage of keeping all information about a single object at a single node. Paths and neighbourhoods are instances of graph objects that may have First-Order Citizenship. A user can explicitly declare a database section as a graph structure, allowing for context specification and encapsulation.
2. Queries can make direct references to a database model's graph structure. The query language algebra has a number of graph-specific operations, such as shortest paths and identifying certain subgraphs. Users are able to build complex searches using declarative graphs and graph operations. In contrast, graph manipulation in logical databases often requires the creation of rather complex rule programs. It is unnecessary to need a complete understanding of the structure in order to formulate valid inquiries. Lastly, it may be useful for browsing purposes to ignore the schema.
3. There are efficient graph algorithms that can be used to do certain tasks, and graph databases may have special ways to store graphs.

3.4 Graph Database Models: Motivations and Applications

The DBG models are inspired by real-world applications where knowledge of the connections between their plots is a distinguishing attribute. The fields of application are split into complex networks and classical networks based on [22]. Classical applications are the operations that prompted the development of multiple GDs, including:

- Classical DB-model generalizations: some criticisms of classical models include their loss of semantics, the flat structure of the data they permit, the user's problems "seeing" the relationship between the data, and the difficulty of modelling complex objects.
- The concept of providing a model in which both data handling and data representations are graph-based was prompted by the fact that graphs have been a key component of the database design process in semantic and object-oriented DB-models.
- For complex applications, the limitations of how languages can be used to express ideas often lead to the creation of models that are closer to these systems.
- Graphical and visual interfaces in addition to geographic, pictorial, and multimedia systems [27].

On the other hand, there are a number of fields known as "complex networks" that have seen the birth of massive data networks sharing certain mathematical features. Recently, the need for database management for certain types of complex networks has been emphasised. Although it is still unclear whether databases can be seen as a single entity, complex networks are divided into four categories:

- **Social networks**: Nodes in social networks are individuals and organizations, whereas connections indicate relationships or flows between nodes. Friendship, commercial partnerships, sexual contact patterns, research networks (collaboration, co-authorship), communication records (mail, phone calls, email), and computer network national security are some examples. In the fields of social network analysis [28], visualizations, and data management in these networks, the amount of work is growing.
- **Information networks**: It describe relationships that show how information flows, like citations among academic articles, the WWW (hyperlinked, hypermedia), peer-to-peer networks [29], and relationships between word classes in a thesaurus.
- **Technological networks**: The structure of technological networks is mostly determined by space and geography. The Internet (as a network of computers), electric power grids, airline routes, telephone networks, and delivery networks are examples of networks (post office). Today, Geographic Information Systems (GIS) cover a large part of this field, which includes roads, railroads, pedestrian traffic, and rivers.
- **Biological networks**: The automation of the data collection process has made it difficult to handle the amount of biological information that biological networks represent. Networks in gene regulation, metabolic pathways, chemical structure, map order, and cross-species homology relationships, for example, exist in the field of genomics. Other biological networks include food webs, neural networks, etc.

3.5 Comparing Graph Database with the Relational Database

A Graph Database is a single-purpose, specialised platform for constructing and managing graphs. Graphs are made up of nodes, edges, and attributes, which are all utilised to represent and store data in a manner that relational databases cannot. On the other hand, a relational database requires a predetermined and carefully designed collection of tables. It is definitely beneficial for storing tabular data that corresponds to a predefined structure, but the interconnections within the data set are weakly accommodated. Thus, forcing a densely connected data set into a relational database causes significant query return time performance problems. Furthermore, the following are some more facts concerning the two databases:

3.5.1 Relational Database

Introduced in the 1970s by E. F. Codd, a database is a software program that enables you to quickly store and retrieve data. Data objects are organised into formal tables in a relational database that can be retrieved and put together in many different ways, like when you want to look up a

person's name. The relational model is shown in Figure 2, along with a few of its terms.

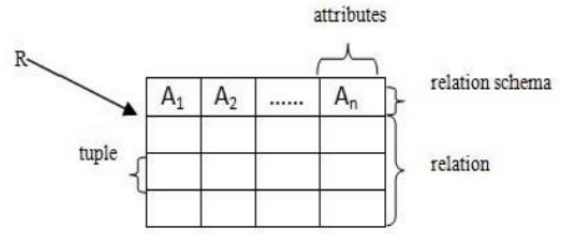

Figure 2: The Relational Model.

A relational database is composed of a series of tables, each with its own unique name. In a table, a row (or tuple) indicates connections between sets of values; the table itself is a collection of these relationships. Since the table strongly matches the mathematical concept of "relation," the term "relational model" was created. Figure 2 displays column headings A1, A2, ...An, which reflect table properties. The conceptual design of the database is defined by the database schema, while a database example is a representation of the data in the database at a specific point. A tuple's characteristics must be distinguishable from those of other tuples, i.e., each tuple must be uniquely recognised. This is accomplished through the use of keys. Extra relationship restrictions can be created using foreign keys. Additionally, more integrity requirements may be provided. Likewise, a query language is necessary for users to query a database [30]. MySQL and Oracle are two of the most widely used relational database tools. With websites, MySQL is becoming increasingly popular. While Oracle is a lightweight system that is incredibly quick, it is mostly utilised for massive database needs in industries such as banking, insurance, Enterprise Resource Planning (ERP), and financial services. Besides this, Oracle is used to handle complicated issues and to support large OLTP settings. There are some differences between them, although both of the tools work in basically the same way [31]. Several advantages of a database designed using the relational model include the following [32]:

- Managing redundancy.
- It is simple to add, update, or delete data.
- Assembling a permanent storage mechanism for software objects.
- Providing a variety of graphical user interfaces.
- It provides data summarizing, retrieval, and reporting capabilities.
- Preventing unauthorized access and enforcing integrity checks.
- Representing intricate relationships between data.

3.5.2 Graph Database (GD)

In mathematical terminology, a graph is made up of two parts: a node (also known as a vertex) and an edge. Each edge symbolises a connection or relationship between two nodes, and each node represents a data object (such as a person, place, item, class, or another piece of information). Relationships are treated as first-class citizens in the graph model. An index-less storage system is one in which comparable items are connected without the need for an index. The physical pointer may also be used to obtain the object's neighbours. In this case, it is a database with CRUD operations that displays a data model built up of graphs like property graphs, superlative graphs, and RDF triads.

The construction of a property graph model is shown in Figure 3, and an example is illustrated in Figure 4. In the graph, the most often used model is the property graph model. In addition, the property graph can run on Neo4j, TigerGraph, AllegroGraph, OrientDB, MongoDB, infiniteGraph, and many other popular graph databases.

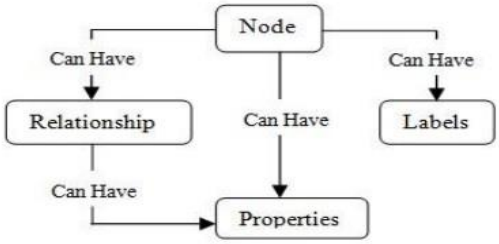

Figure 3: Building blocks for property graph model.

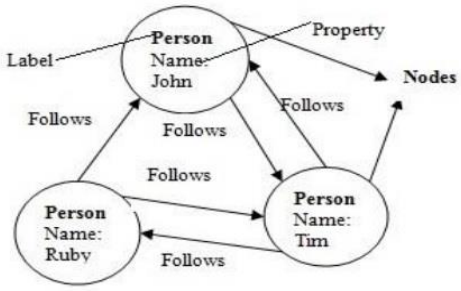

Figure 4: An example of graph model.

4. GRAPH DATABASE TOOLS

4.1 Neo4j (Neo Technology)

Neo4j is a graph database built for network-oriented data, whether in the form of a tree or a general graph, and was initially released in 2007. Neo4j stores data using a network model, where it stores nodes, relationships, and attributes instead of a relational data structure [33], and Figure 5 illustrates an example of Neo4j.

Recently, Neo4j has become one of the top graph databases. A property graph model with nodes, relations, and both having attributes and value pairs are supported by Neo4j. When indexing was first introduced in Neo4j, each index action had to be done directly and manually. Neo4j developed auto indexing, which automatically includes any changes made to data into the index using the global property key, to reduce user participation and enhance the indexing module. The global property key, which has no semantic connection and extracts the same property from several nodes, reduces the accuracy of query responses [15].

Neo4j supports a querying language named Cypher, operating at the highest conceptual database level. Cypher was first developed by Neo Technology for its Neo4j graph database. Neo4j provides complex traversal operations for graphs. The Cypher query language makes querying data from a database quite straightforward. A cipher contains many clauses. MATCH and WHERE are among the most frequently used. These functions vary significantly from those used in SQL. MATCH is used to describe the structure of the pattern being searched for, based mostly on relationships. WHERE is used to apply more constraints to patterns. In addition to clauses for writing, updating, and deleting data, the cipher provides clauses for writing, updating, and erasing data. "Create" and "DELETE" are used to create and delete nodes and relationships [10].

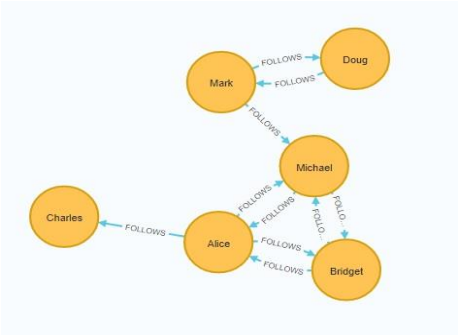

Figure 5: An Example of a Neo4j Graph Database

Furthermore, Neo4j is a graph database that is designed to deal with graphs rather than tables. In comparison to other graph databases, it is a particular kind of graph database. Today, Neo4j is the most popular graph database [6], [34]. Also, Neo4J provides a number of competitive benefits, making it one of the most widely used databases. These are the main features of Neo4j:

- It has its own language, Cypher, which was developed by the corporation specifically for its query techniques. This language is used to manage all of the data in the graph databases.
- Scalability and dependability are two important factors to consider.
- Schema that is adaptable.
- Cypher is a query language developed by Cypher, which is a widely-used and easy-to-read language.
- Its integrity is guaranteed by ACID (Atomicity, Consistency, Isolation, and Durability).
- It offers a user-friendly web interface and APIs, as well as support for a large number of third-party applications.
- Support for Java, Spring, Scala, and JavaScript drivers.
- Backups stored in the cloud;
- Query data may be exported to JSON and XLS formats.
- The world's most active graph community;
- Graph storage and processing are natively supported, resulting in high performance.

Neo4J does not support sharding, and since it is a free version, the community version has problems with the number of nodes, relationships, and attributes. Neo4j utilises the linked list storage format natively and independently to store vertices, edges, and properties. This design often leads to high memory usage and poor performance even when indexes are created.

Real-time graph analysis is required for transactions and operational decisions because it provides a local view of the links between individual data items and allows for action. To learn about the general nature of networks and to simulate the behavior of complex systems, you need graph algorithms that give a fuller overview of patterns and structures across all data and interactions. The Table 1 might assist you in determining the optimal method for your use case.

Table 1: Determine the optimal algorithm

Algorithm Type	Graph Problem	Examples
Path Finding	Determine the most efficient route or evaluate the route's availability and quality	Determine the shortest path between A and B. Routing of telephone calls.
Centrality	Determine the significance of various nodes in a network	Customer segmentation. Identify possible members of a fraud ring.
Community Detection	Evaluating the clustering or partitioning of a group	Identify social media influencers. Identify potential assault targets inside communication and transportation networks.

4.2 TigerGraph

In recent years, TigerGraph has become one of the most prominent distributed graph databases. Its fundamental system is built from the ground up using C++. TigerGraph has great scalability and speed, especially when it comes to hard queries, because it combines native graph storage with MapReduce, highly parallel processing, and fast data compression and decompression. In addition, TigerGraph can be efficiently implemented on a wide number of clusters, and queries may be processed in a distributed manner, enabling it to answer queries on enormously massive graphs that would fail on a single system. Figure 6 shows the main blocks of TigerGraph. It also creates its own sophisticated procedures, such as query language, GSQL. As a proprietary, non-open-source component, TigerGraph is not openly accessible [35]. Geospatial analysis and time series analysis are two of the specific use cases described by TigerGraph. Utilizing the GSQL querying language, TigerGraph is queried. GraphStudio is a web interface that works along with TigerGraph and offers an interface for writing, installing, and visualizing queries; designing and exporting a graph schema; and monitoring database performance [36]

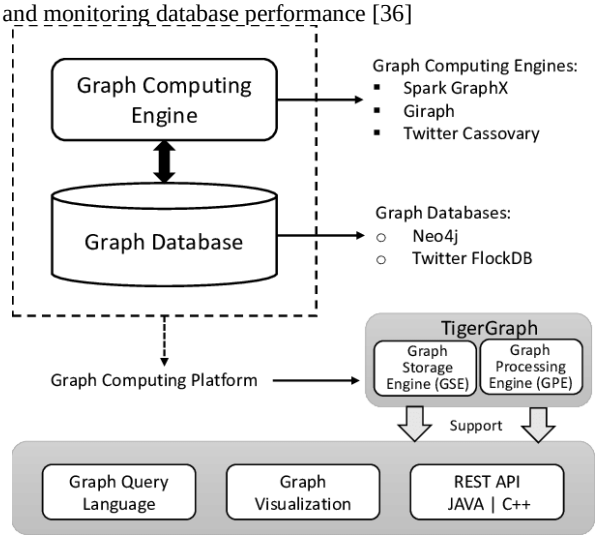

Figure 6: TigerGraph with its main blocks [19].

GSQL is the query language for TigerGraph. GSQL is a straightforward extension of SQL to graph databases, as its name indicates. Before querying, a rigorous schema declaration is enforced. The design employs the labelled property graph data model. It is composed of four parts: the

vertex, the edge, the graph, and the label. The vertex kind is equivalent to a SQL table. It has a name and distinct characteristics. The number of vertices determines the edge type. It may or may not be directed. The number of vertices determines the edge type. It may or may not be directed. In the case of a directed edge, it is possible to declare an optional reverse edge kind. The graph kind specifies the kinds of vertices and edges that compose the graph. The label type only exists to provide compliance with the labeled graph data model. TigerGraph can use the appropriate storage structure and query execution method since everything is determined from the start. GSQL queries are stored procedures that have several SELECT clauses and instructions like branching and looping. GSQL queries resemble SQL stored procedures. This method is prompted by the growing difficulty of some graph computations. Similarly to the MATCH statement in Cypher, the SELECT statement in GSQL matches a route in the graph that begins at a vertex and continues along edges. The FROM clause specifies the route. GSQL adds the "accumulator" (ACCUM) notion associated with a route. On the basis of certain grouping criteria, a path's data may be gathered and combined into accumulators. This is done in parallel, with one process per FROM clause match [20]. So that multi-pass and iterative calculations can be done more easily, the results can be spread out among the vertices. Table 2 shows the basic information about both Neo4j and TigerGraph.

Table 2: Basic information about Neo4j and TigerGraph.

System	Neo4j	TigerGraph
Type	Native	Native
Storage	Linked lists	Native engine
Open Source	Yes	No
Supporting Distributed Processing	No	Yes
Transactional	Yes	Yes
Schema-Free	Yes	No
Implementation Languages	Java	C++
Query Languages	Cypher	GSQL

5. NETWORK CENTRALITY

The concept of centrality was initially used to describe human communication by Bavelas [37], who was interested in the description of communication in small groups of people and assumed a relationship between structural centrality and influence in group processes. Since then, several centrality metrics have been put forward throughout time to measure an individual's significance in a social network [38]. As for graphs, centrality metrics are a vital tool for understanding networks. It aims to determine the significance of a network element based only on the network's structural pattern. The centrality values can be employed to find and evaluate subgoals in multiagent systems and are commonly used as a ranking or identification system [39]. Additionally, centrality metrics are among the most commonly used network-based indicators. The vertex centrality measures have been used in many fields, such as strategic network formation, game theory, social behaviour, transportation, influence and marketing, communication, scientific citation and collaboration, communities, and group problem-solving [40].

Centrality is a crucial feature of complex networks that has a significant impact on the behavior of dynamical processes. A complex network is a graph *G* with a highly structured organization consisting of an ordered pair of disjoint sets (*V, E*), where *V* denotes a set of vertices (or nodes) and *E* is a subset of ordered pairings of distinct elements of *V*, known as edges or arcs. If the network is undirected, meaning that a connection from vertex *i* to vertex *j* also implies a connection from *j* to *i*, the connections are referred to as edges. Otherwise, directed links are referred to as "arcs". Weights can also be assigned to the network edges, indicating the strength or intensity of the relationships between the nodes [41]. In this section, we will define several metrics that will be used to evaluate the performance of both Neo4j and TigerGraph.

5.1 Degree Centrality

Degree Centrality of a node is determined by the number of incident edges that are connected to it (i.e., the number of edges a node has). If the network is directed (edges have direction), then in-degree and out-degree centrality measurements are specified separately. In-degree is the number of edges directed to the node (head endpoints), whereas out-degree is the number of edges directed away from the node (tail endpoints). In these instances, the degree is equal to the sum of the in-degree and out-degree [42]. For normalization, degree centrality can be calculated by:

$$C_D(u) = \frac{K_u}{n-1} \qquad (1)$$

Where *n* represents number of nodes, *K* is the degree of the node *u*, and $C_D(u)$ is the degree centrality of the node *u*.

For an unweighted network, degree centrality has an $O(m)$ time complexity, where *m* is the number of edges. The primary drawback of degree-based centrality is that it only provides local information about a network vertex. In other words, this metric does not account for global structural change. It is clear that this metric is easier and more helpful in a variety of applications [43]. The minimax criteria are commonly employed to determine the optimal locations for emergency facilities such as hospitals, police stations, and military bases, as well as other amenities including schools, gas stations, markets, restaurants, and hotels.

5.2 Betweenness Centrality

One early definition of centrality encapsulating the notion of betweenness was developed from the finding that some nodes, depending on their location in the network, had power over the communication between a pair of other nodes. Nodes that have the ability to communicate between a pair of other nodes have been discovered through laboratory experiments on human interactions. The ability of a node to regulate this communication gives it an important position as a mediator or facilitator. Locally, a node with a high degree has the ability to play this function, depending on the degree of clustering (links) between the node's neighbours, but only for its near neighbours. It does not capture the node's influence on the communication between two distant nodes [43]. The betweenness centrality of node *v* can be calculated by:

$$C_{bet}(v) = \sum_{u,v,w} \frac{\sigma\, uw(v)}{\sigma\, uw} \qquad (2)$$

Where $\sigma\, uw$ is the number of shortest paths between *u* and *w* and $\sigma\, uw(v)$ is the number of shortest paths between *u* and *w* that include v.

5.3 Closeness Centrality

How quickly one may get from one node to every other node in the network is measured by a concept called "closeness centrality". It may be described as the average length of all the shortest routes in a network between a node and every other node. High closeness centrality nodes are significantly closer and can reach other nodes in the network much more rapidly [44]. The closeness centrality of node *v* is defined as:

$$Cl_v = \frac{n-1}{\sum_{u \in V(G)} d(u,v)} \qquad (3)$$

Where Cl_v denotes the closeness centrality of node *v*, *n* is the number of nodes, and $d(u,v)$ is the distance between *u* and *v*.

5.4 Eigenvector Centrality

In a network, a node's effect is quantified by eigenvector centrality. Degree centrality has been extended by eigenvector centrality. According to degree centrality, the total number of connected nodes affects a node's degree centrality. Whereas in eigenvector centrality, the number of adjacent nodes and the significance of the adjacent node are both considered, and all connections are not equal [45].

In general, connections with influential people confer greater influence than connections with less influential people. Not only the connections but also the score (eigenvector centrality) of the connected node are important in eigenvector centrality. Eigenvector centrality is determined by evaluating a node's degree of connectivity with the network's highly connected nodes. A matrix's dominant eigenvector, or eigenvector centrality, is known as an adjacency matrix. In other words, the foundation of Eigenvector Centrality σ_E is the idea that a connection to a more interconnected node helps one's own centrality more than a relationship to a less interconnected node does [46]. So, the σ_E is defined as follows for node *x*:

$$\sigma_E(x) = v_x = \frac{1}{\lambda_{max}(A)} \cdot \sum_{j=1}^{n} a_{jx} \cdot v_j \qquad (4)$$

With $v = (v_1,..., v_n)^T$ referring to an eigenvector for the maximum eigenvalue $\lambda_{max}(A)$ of the adjacency matrix *A*. When employing the Eigenvector centrality method, you should be aware of the following:

- For nodes without any incoming relationships, centrality ratings will converge to 0.
- High-degree nodes have a significant impact on the scores of their neighbours because degree normalization is missing.

5.5 PageRank

PageRank is Google's primary ranking algorithm for web page placement on search engine results pages. PageRank refers to both the system and algorithm that Google employs to rank web pages, as well as the numeric score that is assigned to every page. Using pages as network nodes and links as edges, it modelled human browsing behaviour to rank pages. According to the presumption that a node's significance is the predicted total of the significance of all linked nodes plus the direction of edges, PageRank represents the "importance" of nodes. The probable distributions of random accesses to nodes are represented by their values. PageRank iteratively calculates a normalised and propagated value for each node in a network [47], [48]. Let *x* and *p* be two nodes in a graph *G*; the undirected PageRank of *x* can be calculated as follows:

$$PR(x) = (1 - c) + c \ . \ \sum_{p \in Pnt_{in}(x)} \frac{PR(p)}{|Pnt_{out}(p)|} \qquad (5)$$

$Pnt_{in}(x)$ is the set of nodes pointing to x, $Pnt_{out}(p)$ is the collection of nodes directed by p and $|Pnt_{out}(p)|$ is the cardinality of Pnt_{out}. c is a damping factor with a value in the range [0,1] (usually 0.85). The PageRank works in a directed network, and it keeps figuring out the value of a node based on the PageRanks of the nodes that point to it.

6. DATASET DESCRIPTION

The experiments were done using four datasets belonging to three separate parts, namely, social networks, location-based online social networks, and collaboration networks. These datasets are available as part of the Stanford Large Network Dataset Collection [49]. The following is an explanation of the datasets utilised, with numerical information presented in Table 3:

1. **Social Network**

- **Ego-Facebook**: This dataset includes undirected Facebook "circles" (sometimes known as "friend's lists"). Using this Facebook app, poll respondents' Facebook information was gathered. The dataset consists of node characteristics (profiles), circles, and ego networks. Each user's Facebook internal id has been replaced with a new value, therefore anonymizing Facebook data [50].
- **Musae-Github**: The vast developer social network on GitHub was compiled in June 2019 using data from the open API. Edges are connections between developers who are mutual followers, and nodes are developers who have starred at least 10 repositories. Based on the location, repositories starch, employer, and email address, the vertex characteristics are retrieved [51].

2. **Location-Based Online Social Networks**

- **Brightkite**: This undirected network includes user–user friendship relationships from the previous location-based social network Brightkite, where members disclosed their locations. A node represents a user, and an edge denotes a friendship between the user represented by the node on the left and the user represented by the node on the right.

3. **Collaboration Networks**

- **Ca-HepPh**: The Arxiv HEP-PH (High Energy Physics-Phenomenology) collaboration network is from the e-print arXiv and covers scientific collaborations between authors of papers submitted to the High Energy Physics-Phenomenology category [52].

Table 3: Description of datasets

Network Type	Dataset	Nodes	Edges	Size
Social Networks	Ego-Facebook	4,039	88,234	854.4 KB (854,381 Bytes)
	Musae-github	37,700	289,003	3.3 MB (3,306,148 Bytes)
Location-Based Online	Brightkite	58,228	214,078	2.3 MB (2,277,939 Bytes)
Collaboration Networks	Ca-HepPh	12,008	118,521	2.8 MB (2,783,072 Bytes)

7. EXPERIMENTAL RESULTS

In this section, we present the results obtained by the Neo4j and TigerGraph databases. We used five important centrality metrics as described in Section 5 to show the capabilities of two graph databases that were applied to four different sizes of datasets. Additionally, both are implemented using Linux (Ubuntu 20.04.4 LTS, 64-bit) on a Core i7-5600U, 2.60 GHz x 4 CPU, with 16 GB of physical memory, and standard Solid State Drives (SSD). Then, in Neo4j, we use the Neo4j Browser (4.4.5) and Neo4j Desktop (1.4.15) to execute the used metrics, while in TigerGraph, we use GraphStudio. Besides, the results that are shown in Figure 1 are the findings that were obtained by running each query five times and then determining the average of those five runs in order to get the execution rate for each of the centrality metrics. The data loading experiments are executed once because they require a much longer time, especially in Neo4j.

7.1 Loading the Dataset

To import data, both Neo4j and TigerGraph need a CSV file, and TigerGraph's results are much better than those of Neo4j during the data loading phase, as shown in Figure 7. For all datasets, the data import time of TigerGraph is shorter than that of Neo4j. This means that TigerGraph has a faster loading time than Neo4j. In other words, TigerGraph has an advantage

over Neo4j in the loading phase. In contrast, Neo4j requires more time to load the data and for the data to become available for use. Neo4j is very user-friendly with importing data compared to TigerGraph. In Neo4j, you can specify the exact amount of data to be imported, whereas in TigerGraph, you can only provide a percentage of data.

TigerGraph is very efficient, even when loading large datasets. As can be seen from Figure 7, Musae-Github is much bigger than Ego-Facebook; however, the loading time for Musae-Github is only two times that of Ego-Facebook. Regarding Neo4j, there is a big difference when loading a large dataset versus a small dataset, as can be seen again in Figure 7.

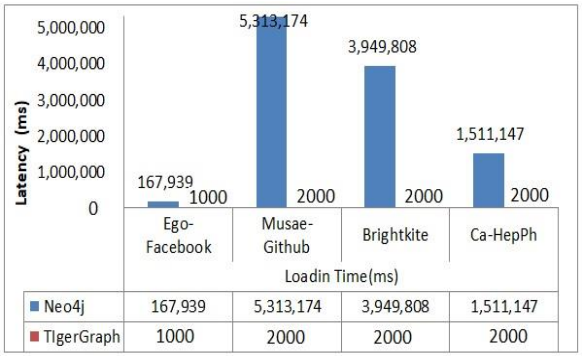

Figure 7: Shows the loading time for Neo4j and TigerGraph(ms)

7.2 Evaluation and results

7.2.1 Neo4j and TigerGraph: In this part of the article, we conducted an evaluation of the two graph databases that were used.

Neo4j: Figure 8 and Table 4 display the latency time of Neo4j for each centrality metric. As can be seen, Neo4j performs very efficiently in calculating centrality metrics. One of the most important features of Neo4j is that when importing data, it first stores the data in memory to improve performance and then indexes the data until it is ready for use, at which point queries can be applied to the data. Therefore, if we look at the results of implementing the metrics in Table 4 and Figure 8, we see that the degree, eigenvector, and PageRank did not take much time; only betweenness and closeness took more time. This is due to the complexity and type of algorithm implemented in Neo4j. This means that the worst among them is the betweenness, followed by the closeness, which takes longer than the others, but for the rest, there is a small difference between them.

Table 4: Performance of Neo4j for centrality metrics.

Metrics	Latency (ms)			
	Ego-Facebook	Musae-github	Brightkite	Ca-HepPh
Degree Centrality	54	238	322	112
Betweenness Centrality	7,020	629,833	756,921	56,718
Closeness Centrality	249	17,958	24,315	3,699
Eigenvector Centrality	163	733	701	487
PageRank	168	609	626	447

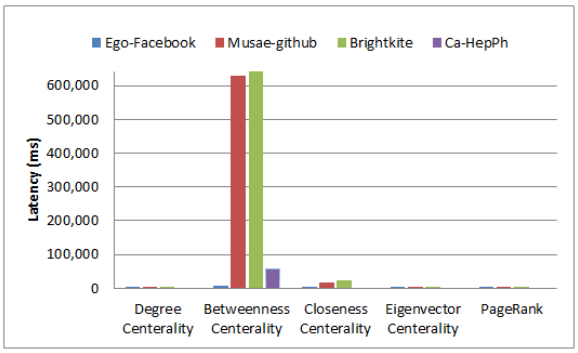

Figure 8: Shows the latency time of Neo4j for centrality metrics.

TigerGraph: If we compare the results obtained by TigerGraph for each metric in Table 5 with those obtained by Neo4j in Table 4, we can see that they are completely different and larger than Neo4j. See Figure 9 and Table 5 for more details. The datasets that were used have quite a large variation when it comes to latency time. The degree centrality is the best metric applied to all datasets, while the betweenness centrality is the worst. This is indeed because of their complexity and the algorithm implemented in TigerGraph. Betweenness requires calculating all the shortest paths from the node that you find its betweenness with all the others and the number of those nodes that pass through the node that you find its betweenness, and then dividing them. As for closeness, it comes after the betweenness in the term of latency. This also takes time when centrality metrics are applied to it, but less than betweenness centrality. For instance, if we look at Table 5, the latency of the closeness of the Brightkite dataset is equal to 781,264 ms, while the betweenness is much bigger and is equal to 3,294,742 ms.

In reality, dealing with a TigerGraph interface takes time because of the need to design schema, map data to graph, load graph, explore graph, and write queries sequentially so that a query can be executed. Each of these requires different phases of implementation. One other point is that when writing queries, you must first debug the errors if any, and then save the query. After that, the query must be installed to store the query inside the database, and then, it can be executed. All the points mentioned above require a lot of time to be ready for use, which means that dealing with the TigerGraph interface costs a lot of time until certain queries can be applied.

Table 5: Performance of TigerGraph for centrality metrics.

Metrics	Latency (ms)			
	Ego-Faceboo k	Musae-Github	Brightkit e	Ca-HepPh
Degree Centrality	59	305	252	168
Betweennes s Centrality	183,007	1,644,11 7	3,294,74 2	1,510,13 8
Closeness Centrality	15,001	526,565	781,264	89,314
Eigenvector Centrality	2,049	17,855	13,057	2,548
PageRank	1,413	5,975	3,839	2,468

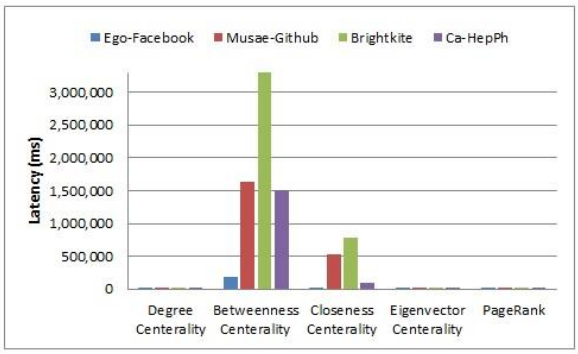

Figure 9: Shows the latency time of TigerGraph for centrality metrics.

7.2.2 The Overall Evaluations: In this section, we provide an overall evaluation to demonstrate the capabilities of Neo4j and TigerGraph based on the centrality metrics used in Section 5. This evaluation is based on the results of the data presented in Table 6. The findings can be summarised as follows:

TigerGraph performs significantly better than Neo4j during the data loading phase, particularly with large datasets. There is very little difference in loading time when you are comparing the loading times of small and large datasets in TigerGraph. When looking at the results achieved by the Neo4j in the data loading stage, there is a big difference compared to the TigerGraph. The Neo4j database takes a lot of time, especially when the dataset is large. There is a very big difference in the loading time of the large datasets compared to the small datasets. TigerGraph achieved lower loading phase latency than Neo4j for each dataset, as shown in Figure 7. For instance, to load the Musae-Github dataset in Neo4j, it took 5,313,174 ms; however, for TigerGraph, it took only 2000 ms.

In terms of centrality metrics, the performance of Neo4j outperforms TigerGraph. This is because Neo4j first stores the data in memory to improve performance before allowing queries to be applied to it. In contrast, the TigerGraph did not have such a feature, so when implementing metrics, it takes a lot of time, especially when the datasets are large. This is because TigerGraph employs a hybrid memory and disk storage model during query execution. Also, the results of this feature are quite clear in Table 6. If we look at Table 6, we can see that the results of the latency of each metric gained by Neo4j are much lower than those obtained by TigerGraph. In the degree centrality the difference between Neo4j and the TigerGraph is small; for example, if we look at the Ego-Facebook dataset, the latency of Neo4j is equal to 54, whereas in the TigerGraph it is equal to 59. However, for the betweenness centrality, there is a significant difference in latency between the two commonly used graph databases. For instance, if we look at the same dataset, the latency of the betweenness centrality that was achieved by the TigerGraph is equal to 183,007 ms, but the result that was gained by the Neo4j is equal to 7,020 ms, which is much less than what was achieved by the TigerGraph.

In fact, when we deal with the interfaces of the two graph databases, Neo4j is the easiest compared to TigerGraph. Because in TigerGraph, in order for the query to be ready for execution, TigerGraph requires many stages before starting to execute the query, and these operations are time-consuming, TigerGraph takes a long time for the query to be ready to be executed. With Neo4j, you don't need to go through any other steps before executing the query.

Table 6: Performance of Neo4j and TigerGraph for centrality metrics.

Metrics	Latency (ms)							
	Ego-Facebook		Musae-Github		Brightkite		Ca-HepPh	
	Neo4j	TigerGraph	Neo4j	TigerGraph	Neo4j	TigerGraph	Neo4j	TigerGraph
Degree Centrality	**54**	59	**238**	305	**322**	347	**112**	168
Betweenness Centrality	**7,020**	183,007	**629,833**	1,644,117	**756,921**	3,294,742	**56,718**	1,510,138
Closeness Centrality	**249**	15,001	**17,958**	526,565	**24,315**	781,264	**3,699**	89,314
Eigenvector Centrality	**163**	2,049	**733**	17,855	**701**	13,057	**487**	2,548
PageRank	**168**	1,413	**609**	5,975	**626**	3,839	**447**	2,468

8. CONCLUSION

Graph databases can manage complex and massive amounts of data without requiring a restructuring since only the relationships between the nodes need to be added. In addition, graph databases offer index-free adjacency results that only search for entries that are closely connected to other records, eliminating the need for an index and accelerating retrieval speeds. In this paper, we experimentally evaluate the performance of Neo4j and TigerGraph graph databases for centrality metrics including degree, betweenness, closeness, eigenvector, and PageRank centralities. Additionally, we also evaluated the performance of data loading for both graph databases. Experimental results showed that Neo4j achieved excellent performance over TigerGraph for computing those metrics. However, the performance of data loading in TigerGraph is promising compared to Neo4j. In the future, we aim to apply some other metrics and algorithms to the two graph databases utilised in this study in order to examine their capabilities in more depth and also compare them with some other graph databases.

REFERENCES

D. Fernandes and J. Bernardino, “Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB.,” in *Data*, 2018, pp. 373–380.

F. Chen, Y.-C. Wang, B. Wang, and C.-C. J. Kuo, “Graph representation learning: a survey,” *APSIPA Trans. Signal Inf. Process.*, vol. 9, 2020.

N. Shadbolt, T. Berners-Lee, and W. Hall, “The semantic web revisited,” *IEEE Intell. Syst.*, vol. 21, no. 3, pp. 96–101, 2006.

X.-M. Xu, J. Zhan, and H. Zhu, “Using social networks to organize researcher community,” in *International Conference on Intelligence and Security Informatics*, 2008, pp. 421–427.

R. Soussi, M.-A. Aufaure, and H. Baazaoui, “Graph database for collaborative communities,” in *Community-Built Databases*, Springer, 2011, pp. 205–234.

R. Kumar Kaliyar, “Graph databases: A survey,” in *International Conference on Computing, Communication & Automation*, 2015, pp. 785–790.

B. Ristevski, “Using Graph Databases for Querying and Network Analysing,” 2019.

A. G. Baset, “Graphical Database Architecture For Clinical Trials,” 2015.

R. Angles and C. Gutierrez, “An introduction to graph data management,” in *Graph Data Management*, Springer, 2018, pp. 1–32.

H. Lu, Z. Hong, and M. Shi, “Analysis of film data based on Neo4j,” in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, 2017, pp. 675–677.

S. Almabdy, “Comparative analysis of relational and graph databases for social networks,” in *2018 1st International Conference on Computer Applications & Information Security (ICCAIS)*, 2018, pp. 1–4.

A. Lysenko, I. A. Roznovăţ, M. Saqi, A. Mazein, C. J. Rawlings, and C. Auffray, “Representing and querying disease networks using graph databases,” *BioData Min.*, vol. 9, no. 1, pp. 1–19, 2016.

M. Šestak, M. Heričko, T. W. Družovec, and M. Turkanović, “Applying k-vertex cardinality constraints on a Neo4j graph database,” *Future Gener. Comput. Syst.*, vol. 115, pp. 459–474, 2021.

M. Macak, M. Stovcik, B. Buhnova, and M. Merjavy, “How well a multi-model database performs against its single-model variants: Benchmarking OrientDB with Neo4j and MongoDB,” in *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, 2020, pp. 463–470.

M. Desai, R. G. Mehta, and D. P. Rana, “An empirical analysis to identify the effect of indexing on influence detection using graph databases,” *Int J Innov Technol Explor. Eng*, vol. 8, pp. 414–421, 2019.

S. Beis, S. Papadopoulos, and Y. Kompatsiaris, “Benchmarking graph databases on the problem of community detection,” in *New Trends in Database and Information Systems II*, Springer, 2015, pp. 3–14.

R. Wang, Z. Yang, W. Zhang, and X. Lin, “An empirical study on recent graph database systems,” in *International Conference on Knowledge Science, Engineering and Management*, 2020, pp. 328–340.

C. Liu and H. Duan, “A Graph Database Storage Engine for Provenance Graphs,” *DBKDA 2020*, p. 8.

T. Chen, C. Yuan, G. Liu, and R. Dai, “Graph based platform for electricity market study, education and training,” in *2018 IEEE Power & Energy Society General Meeting (PESGM)*, 2018, pp. 1–5.

F. Rusu and Z. Huang, “In-depth benchmarking of graph database systems with the Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB),” *ArXiv Prepr. ArXiv190707405*, 2019.

A. Deutsch, Y. Xu, M. Wu, and V. E. Lee, “Aggregation support for modern graph analytics in tigergraph,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 377–392.

R. Angles and C. Gutierrez, “Survey of graph database models,” *ACM Comput. Surv. CSUR*, vol. 40, no. 1, pp. 1–39, 2008.

N. S. Patil, P. Kiran, N. P. Kiran, and N. P. KM, “A survey on graph database management techniques for huge unstructured data,” *Int. J. Electr. Comput. Eng.*, vol. 8, no. 2, p. 1140, 2018.

R. Angles, “The Property Graph Database Model.,” in *AMW*, 2018.

A. Das, A. Mitra, S. N. Bhagat, and S. Paul, “Issues and Concepts of Graph Database and a Comparative Analysis on list of Graph Database tools,” in *2020 International Conference on Computer Communication and Informatics (ICCCI)*, 2020, pp. 1–6.

R. H. Güting, “GraphDB: Modeling and querying graphs in databases,” in *VLDB*, 1994, vol. 94, pp. 12–15.

M. Consens and A. Mendelzon, “Hy+ A hygraph-based query and visualization system,” *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 511–516, 1993.

U. Brandes, *Network analysis: methodological foundations*, vol. 3418. Springer Science & Business Media, 2005.

W. Nejdl, W. Siberski, and M. Sintek, “Design issues and challenges for RDF-and schema-based peer-to-peer systems,” *ACM SIGMOD Rec.*, vol. 32, no. 3, pp. 41–46, 2003.

H. R. Vyawahare, P. P. Karde, and V. M. Thakare, “A hybrid database approach using graph and relational database,” in *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*, 2018, pp. 1–4.

K. Islam, K. Ahsan, S. A. K. Bari, M. Saeed, and S. A. Ali, “Huge and Real-Time Database Systems: A Comparative Study and Review for SQL Server 2016, Oracle 12c & MySQL 5.7 for Personal Computer,” *J. Basic Appl. Sci.*, vol. 13, pp. 481–490, 2017.

N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, “A survey and comparison of relational and non-relational database,” *Int. J. Eng. Res. Technol.*, vol. 1, no. 6, pp. 1–5, 2012.

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*, 2010, pp. 37–48.

J. Guia, V. Gonçalves Soares, and J. Bernardino, “Graph Databases: Neo4j Analysis:,” in *Proceedings of the 19th International Conference on Enterprise Information Systems*, Porto, Portugal, 2017, pp. 351–356. doi: 10.5220/0006356003510356.

G. Li, H. T. Shen, Y. Yuan, X. Wang, H. Liu, and X. Zhao, *Knowledge Science, Engineering and Management: 13th International Conference, KSEM 2020, Hangzhou, China, August 28-30, 2020, Proceedings, Part I*, vol. 12274. Springer Nature, 2020.

S. Baker Effendi, B. van der Merwe, and W.-T. Balke, “Suitability of graph database technology for the analysis of spatio-temporal data,” *Future Internet*, vol. 12, no. 5, p. 78, 2020.

A. Bavelas, “A mathematical model for group structures,” *Hum. Organ.*, vol. 7, no. 3, pp. 16–30, 1948.

V. Latora and M. Marchiori, “A measure of centrality based on network efficiency,” *New J. Phys.*, vol. 9, no. 6, p. 188, 2007.

S. Adali, X. Lu, and M. Magdon-Ismail, “Deconstructing centrality: thinking locally and ranking globally in networks,” in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2013, pp. 418–425.

Ö. Şimşek and A. Barto, “Skill characterization based on betweenness,” *Adv. Neural Inf. Process. Syst.*, vol. 21, 2008.

F. A. Rodrigues, “Network centrality: an introduction,” in *A mathematical modeling approach from nonlinear dynamics to complex systems*, Springer, 2019, pp. 177–196.

A. Saxena and S. Iyengar, “Centrality measures in complex networks: A survey,” *ArXiv Prepr. ArXiv201107190*, 2020.

K. Das, S. Samanta, and M. Pal, “Study on centrality measures in social networks: a survey,” *Soc. Netw. Anal. Min.*, vol. 8, no. 1, pp. 1–11, 2018.

P. Choudhary and U. Singh, “A survey on social network analysis for counter-terrorism,” *Int. J. Comput. Appl.*, vol. 112, no. 9, pp. 24–29, 2015.

A. Bihari and M. K. Pandia, “Eigenvector centrality and its application in research professionals’ relationship network,” in *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, 2015, pp. 510–514.

A. Landherr, B. Friedl, and J. Heidemann, “A critical review of centrality measures in social networks,” *Bus. Inf. Syst. Eng.*, vol. 2, no. 6, pp. 371–385, 2010.

K. Henni, N. Mezghani, and C. Gouin-Vallerand, “Unsupervised graph-based feature selection via subspace and pagerank centrality,” *Expert Syst. Appl.*, vol. 114, pp. 46–53, 2018.

A. Hashemi, M. B. Dowlatshahi, and H. Nezamabadi-Pour, “MGFS: A multi-label graph-based feature selection algorithm via PageRank centrality,” *Expert Syst. Appl.*, vol. 142, p. 113024, 2020.

“Stanford Large Network Dataset Collection (SNAP).” Accessed: Feb. 08, 2022. [Online]. Available: “Stanford large network dataset,” http: //snap.stanford.edu/data/index.html.

J. Leskovec and J. Mcauley, “Learning to discover social circles in ego networks,” *Adv. Neural Inf. Process. Syst.*, vol. 25, 2012.

B. Rozemberczki, C. Allen, and R. Sarkar, “Multi-scale attributed node embedding (2019),” *ArXiv Prepr. ArXiv190913021*, 2019.

J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification and shrinking diameters,” *ACM Trans. Knowl. Discov. Data TKDD*, vol. 1, no. 1, pp. 2-es, 2007.