|  |  |
| --- | --- |
| **Course code and name:** | MASTER’S PROJECT – F21MP |
| **Type of assessment:** | **Individual** |
| **Coursework Title:** | MONGODB VS NEO4J (COMPARISION OF DATABASE TECHNOLOGIES USING BIOMEDICAL DATASET) |
| **Student Name:** | VINEETH RAJA BANALA |
| **Student ID Number:** | H00396398 |

|  |
| --- |
| **Declaration of authorship. By signing this form:**   * **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section. * I confirm that I have read, understood and followed the University’s Regulations on plagiarism as published on the [University’s website](https://www.hw.ac.uk/uk/students/studies/examinations/plagiarism.htm), and that I am aware of the penalties that I will face should I not adhere to the University Regulations. * I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on [Academic Integrity and Plagiarism](https://heriotwatt.sharepoint.com/sites/skillshub/SitePages/Academic-Integrity-and-Plagiarism.aspx)   **Student Signature***: VINEETH RAJA BANALA*  **Date**: *14/08/2024* |

Copy this page and insert it into your coursework file in front of your title page.

For group assessment each group member must sign a separate form and all forms must be included with the group submission.

**Your work will not be marked if a signed copy of this form is not included with your submission.**

RESEARCH PROJECT

TITLE: MONGODB VS NEO4J (COMPARISION OF DATABASE TECHNOLOGIES USING BIOMEDICAL DATASET)

(Academic Year 2024-25)

COURSECODE: F21MP

BY: VINEETH RAJA BANALA (H00396398)

SUPERVISOR: PROFESSOR ALBERT BURGER

School of Mathematical and Computer Science

Heriot Watt University

ABSTRACT:

This research report conducts a comparative analysis of MongoDB and Neo4j, two prominent database technologies, by using biomedical dataset. While existing studies often explored specific aspects of these database technologies, there is significant gap remains in comparative analysis in aspects like performance, scalability, suitability. Our research report proposes to address this gap by using systematic evaluation methodology. we aim to provide useful relative information regarding the strengths and weaknesses of MongoDB an Neo4j ,so user can make informed decisions for particular use case. Preliminary analysis indicates that MongoDB is more efficient in handling semi structured data with high throughout, contracting Neo4j is better at querying highly connected data structures. Detailed results and implications for various use cases will be presented in the final report.

The research outlines a structured approach that includes a literature review, methodology, comparative analysis, and discussion of the results. First, we will review existing literature to understand the database technologies in related to biomedical data, then through systematic benchmarking process that compare database technologies to predetermined factors like performance, scalability, suitability for different use case. This comparative analysis will provide insights to which database is more suitable for unique characteristics of biomedical data. We aim to provide valuable insights to researchers and developers for selecting more suitable database technology for their specific biomedical application, by evaluating MongoDB and Neo4j with biomedical dataset.

TABLE OF CONTENTS

[ABSTRACT: 2](#_Toc174474357)

[Chapter – 1 INTRODUCTION 5](#_Toc174474358)

[1.1 Goal 5](#_Toc174474359)

[1.2 Motivation 5](#_Toc174474360)

[1.3 Previous Work 6](#_Toc174474361)

[1.4 Evaluation Method 6](#_Toc174474362)

[1.5 Expected Results 6](#_Toc174474363)

[Chapter – 2 LITERATURE REVIEW 7](#_Toc174474364)

[2.1. Databases and their evolution 7](#_Toc174474365)

[2.1.1 Hierarchical and Network Databases 8](#_Toc174474366)

[2.2.2 Relational Databases 8](#_Toc174474367)

[2.3 NoSQL Databases 8](#_Toc174474368)

[2.3.1 Different Types of NoSQL Databases 9](#_Toc174474369)

[2.4 Graph Databases 12](#_Toc174474370)

[2.5 MongoDB and Its Applications in Biomedical Datasets 15](#_Toc174474371)

[2.6 Neo4j and Its Applications in Biomedical Datasets 16](#_Toc174474372)

[2.7 Comparative Analysis of MongoDB and Neo4j: 17](#_Toc174474373)

[Chapter – 3 REQUIREMENT ANALYSIS 18](#_Toc174474374)

[3.1 Functional Requirements 18](#_Toc174474375)

[3.1.1 Data Handling 18](#_Toc174474376)

[3.1.2 Performance Measurement: 18](#_Toc174474377)

[3.1.3 Query Generation: 19](#_Toc174474378)

[3.2 Non-Functional Requirements: 19](#_Toc174474379)

[3.2.1 Performance: 19](#_Toc174474380)

[3.2.2 Usability: 19](#_Toc174474381)

[3.2.3 Reliability: 19](#_Toc174474382)

[Chapter – 4 IMPLEMENTATION 20](#_Toc174474383)

[4.1 Development Environment & Language 20](#_Toc174474384)

[4.1.1 Google Colab 20](#_Toc174474385)

[4.1.2 Python 21](#_Toc174474386)

[4.2 Dataset Overview 21](#_Toc174474387)

[4.3 Databases Configuration 22](#_Toc174474388)

[4.3.1 MongoDB Connection Setup 22](#_Toc174474389)

[4.3.2 Neo4j Connection Setup 23](#_Toc174474390)

[4.4 Data Insertion to Databases 24](#_Toc174474391)

[4.5 Comparative Queries Implementation 26](#_Toc174474392)

[4.5.1 Data Fetching 26](#_Toc174474393)

[4.5.2 Indexing impact 33](#_Toc174474394)

[4.5.3 Shortest path: 41](#_Toc174474395)

[4.5.4 Normalization 43](#_Toc174474396)

[4.5.5 Schema Flexibility 46](#_Toc174474397)

[4.6 Latency determination 48](#_Toc174474398)

[Chapter – 5 EVALUATION AND RESULTS 50](#_Toc174474399)

[5.1 Latency 51](#_Toc174474400)

[5.1.1 Latency outcomes 51](#_Toc174474401)

[5.1.3 Indexing impact 56](#_Toc174474402)

[5.1.4 Shortest Path Calculation 59](#_Toc174474403)

[5.1.5 Normalization 61](#_Toc174474404)

[5.1.6 Schema flexibility 63](#_Toc174474405)

[5.2 Conclusion 65](#_Toc174474406)

[5.3 Future Work 66](#_Toc174474407)

[REFERENCES: 67](#_Toc174474408)

# Chapter – 1 INTRODUCTION

In today’s digital era, data has become an important part of decision making and innovation across many industries. With the increasing complexity of data structures, selecting an appropriate database management system (DBMS) is crucial to organizations which are aiming to harness the full potential of their data assets. MongoDB and Neo4j are two prominent DBMS options, each has special features and capabilities suited for specific use cases. By conducting comparative analysis of these database technologies, the main objective of this research is to provide valuable insights for researchers or developers in making informed decisions regarding their database technologies choices.

## 1.1 Goal

## This project aims at performing a comprehensive comparison of MongoDB and Neo4j in terms of their performance. The areas of interest are response time of queries, memory usage, and how well the system performs certain operations on the data, for instance, grouping, sorting, and finding shortest paths. By analyzing these aspects systematically, the project will identify the merits and demerits of each of the database technologies and provide recommendation on the usefulness of the technologies in various contexts.

## 1.2 Motivation

## The motivation for this project arises from the growing need to optimize data management systems in environments where both the volume and complexity of data are rapidly increasing. MongoDB has a document-oriented data model to handle unstructured data and Neo4j has a graph data model to handle relationships within data. It is important for the developers and researchers who are involved in the implementation of the database technologies to know how they operate under different conditions.

## 1.3 Previous Work

Prior works have considered the two systems separately and analyzed their performance in terms of features, for example, query optimization. However, there are relatively few studies that have systematically compared these two technologies in terms of their performance measures with real data sets. This work extends prior work by offering a more comprehensive assessment, especially when it comes to query execution and data processing.

## 1.4 Evaluation Method

The performance evaluation of this research involves comparison of the performance of the proposed system against a set of standard benchmarks concerning different query operations such as data retrieval, aggregation and sorting, and shortest path finding, index impact. Performance indicators like query response time, number of operations per second, and memory usage are used to evaluate MongoDB and Neo4j in various cases.

## 1.5 Expected Results

The expected outcomes of this project include the following: The comparison of the performance of MongoDB and Neo4j especially in handling of complex data operations. The research is expected to identify the conditions under which each DBMS is optimal and the conditions under which they are suboptimal. These findings will be useful for developers and organisations to make the right choice on the type of database technology to apply depending on the need at hand, hence better ways on how to manage data.

# Chapter – 2 LITERATURE REVIEW

The Literature Review holds a prominent indenture in aiding research with an extensive and detailed evaluation of the extant reported research findings and the controversial views related to the selected area of inquiry. Within the FIDEC of this dissertation, this attention shifts its focus on the evaluation and analysis between MongoDB and Neo4j in the context of biomedical datasets. The importance of the review is stressed by the growing complexity of data in biomedicine, which needs systems of databases that can give auto-speed for storage, retrieval, and analysis.

MongoDB is a document-based NoSQL database that stands out for being highly agile and scalable - accounts for all the flexibility required to work with 'big' data in biomedical informatics. However, in the sense of Neo4j, a graph database, also called a graph database, its most significant advantage understands the interrelation of data, which is the first place in the biomedical field where it has relations. Doing this by a systematic literature review will clarify their strengths as well as deficiencies on an individual level and compared with each other, which in turn can facilitate the identification of the most suitable for colonizing the front-end part of the client-server application using React.js with the Spring Boot framework at backend. The underlying objective is to contribute to efficient biomedical data management and, consequently, the effects on research and clinical outcomes.

## 2.1. Databases and their evolution

The evolution of databases reflects a journey from early data management systems to complex, modern solutions that address diverse and dynamic requirements. This evolution can be divided into distinct phases, each marked by technological advancements and shifts in data management paradigms.

### 2.1.1 Hierarchical and Network Databases

The first database systems were designed in the 1960s and they included the hierarchical and the network type of databases. Another type of database was the hierarchical one, that was developed to work with data of a tree-like nature, for example IMS of International Business Machines. In this model, data is organized in parent-child relationships, which is good when applications have a fixed schema, and the data structure is known in advance. The hierarchical model, which is effective for some types of searches, had certain problems with flexibility and data access and manipulation in more complicated cases (Date, 2004; Smith, 2012).

Network databases, which appeared slightly later, offered a more liberal concept. The CODASYL database model provided for more data record relationships by supporting multiple data record connections. This model eliminated some of the problems encountered with the hierarchical databases, but it also had problems of schema change and efficient search (CODASYL, 1971; Codd, 1979).

### 2.2.2 Relational Databases

The relational model developed by Edgar F. Codd in 1970 shifted the approach of database management where data is stored in the form of tables (Codd, 1970). Another model was data centric where data was made independent, normalized and queried and controlled using SQL. Relational approach was more flexible and powerful in managing data relations and queries, which led to the widespread of this approach (Date, 2004; Date & Darwen, 2000).

Relational databases were soon adopted by the industry and many systems were developed such as IBM DB2, Oracle, and Microsoft SQL Server. These databases offered strong characteristics like the ACID transaction, data integrity, and support for various kinds of queries and thereby useful for almost all sorts of applications inclusive of a financial system, enterprise resource planning, and many more (Silberschatz, Korth, & Sudarshan, 2010; Elmasri & Navathe, 2015).

## 2.3 NoSQL Databases

NoSQL databases as the new type of data storage can be considered as the new paradigm due to the problems that are inherent in relational models. Towards the beginning of the 2000s, the new possibilities offered by internet, social networks and mobile applications caused a significant increase in the data volume and data processing intricacy. This increase in data brought new problems for relational DBMSs. Although still being very solid, reliable, and efficient, they began to show limitations in managing large-scale, unstructured and semi-structured data (Stonebraker & Cattell, 2011).

For the relational databases, since the schema and data structures are rigid, they posed a big problem as data increased at an exponential rate. The conventional way of obtaining better solutions was to scale up, that is, acquire more powerful hardware, but this was now becoming ineffective and expensive. Moreover, the properties of ACID (Atomicity, Consistency, Isolation, Durability) that were beneficial in the case of relational systems sometimes conflicted with the need for high availability and performance in distributed systems (Sadalage & Fowler, 2012).

In turn, with the help of new possibilities, new problems appeared, and, in response to these changes, NoSQL databases appeared as an option to the relational model. The term NoSQL, which is an acronym for ‘Not Only SQL’, is an umbrella term that refers to many database technologies that have been specifically built for handling different types of data and for fast and large-scale operations. NoSQL is designed to be flexible, scalable and high performing and it does not follow the same rigor of ACID compliance as the traditional relational databases do (Cattell, 2011).

The NoSQL movement was defined by the creation of several technologies, each of which solved certain problems. Document-oriented databases, key-value stores, column-family stores, and graph databases appeared to address various problems of data management. These technologies enabled organizations to grow in the horizontal plane as data was spread across various servers making it easier to manage large volumes of information (Stonebraker & Cattell, 2011).

NoSQL also brought focus on operational aspects of data, lack of schema with it meant less focus on upfront data modeling which meant that applications could be developed and changed quickly. This flexibility allowed organizations to scale to the changes in needs and work with semi-structured or unstructured data, which could not be easily stored in the relational tables (Sadalage & Fowler, 2012).

In summary, the development of NoSQL database was a reaction to the modern application requirements that were not met by traditional relational databases, as well as it provided solutions for the utilization of vast, various types of data (Meyer et al., 2015).

### 2.3.1 Different Types of NoSQL Databases

***Key-Value Stores:***

One of the simplest and the most primitive forms of NoSQL databases are key-value stores. In this model, data is only in the form of collection of keys and values where the key is distinct from other keys, but the value may not necessarily be unique. This structure is very effective when there are many retrieval and store operations to be performed in the shortest time possible. Kleppmann (2017) also points that KVS is suitable for caching and session management as this kind of data store is fast and provides little latency. For instance, Redis, an in-memory key-value store, is famous for being very fast in data retrieval and modification and is thus suitable for purposes that demand real-time data processing (Carlson, 2013). Redis includes such types of data as strings, lists, sets, and hashes which make it even more useful and fast.

On the other hand, Amazon DynamoDB, which is a highly available and fault-tolerant–based key value store is designed for large scale applications. According to DeCandia et al. (2007), DynamoDB uses consistent hashing and replication to store data across the nodes, and hence it is highly available and scalable. A result of the key–value model’s relative simplicity is that horizontal scaling is relatively straightforward, which is highly important for applications that are growing quickly and may have unpredictable load patterns (Jiang et al. , 2014).

***Column-Family Stores:***

One of the simplest and the most primitive forms of NoSQL databases are key-value stores. In this model, data is only in the form of collection of keys and values where the key is distinct from other keys, but the value may not necessarily be unique. This structure is very effective when there are many retrieval and store operations to be performed in the shortest time possible. Kleppmann (2017) also points that KVS is suitable for caching and session management as this kind of data store is fast and provides little latency. For instance, Redis, an in-memory key-value store, is famous for being very fast in data retrieval and modification and is thus suitable for purposes that demand real-time data processing (Carlson, 2013). Redis includes such types of data as strings, lists, sets, and hashes which make it even more useful and fast.

On the other hand, Amazon DynamoDB, which is a highly available and fault-tolerant–based key value store is designed for large scale applications. According to DeCandia et al. (2007), DynamoDB uses consistent hashing and replication to store data across the nodes, and hence it is highly available and scalable. A result of the key–value model’s relative simplicity is that horizontal scaling is relatively straightforward, which is highly important for applications that are growing quickly and may have unpredictable load patterns (Jiang et al., 2014).

***Document Stores:***

Document stores, which is a category of NoSQL databases, is used to store, manage and process document-oriented data. They are best suited when the demands for data are about equally divided and have features such as scalability and simplicity of implementation. Document oriented databases are commonly used to store data in JSON, BSON, or XML form, and there is no fixed document structure or data type as documents can be of different types and may contain fields that are different from other documents. This flexibility is a major strength in today’s application development where virtually any kind of data and structures can be handled.

***MongoDB:***

MongoDB is another well-known document-based database that enjoys a high popularity and is distinguished by its versatility and the presence of a wide range of features. MongoDB saves data in the Binary JSON format, which is an extension of JSON that adds new data types and optimizations (MongoDB, 2024). It supports collections which have a dynamic schema, this means that the structure of documents can be changed by the developers without impacting the rest of the documents in the collection. This schema flexibility is of advantage for applications where the data model must change over time (MongoDB, 2024). MongoDB supports versatile querying and indexing that enhances the retrieval of data and performance (MongoDB, 2024). Also, MongoDB supports replication and sharding, which make the system horizontally scalable and provide high availability and the possibility of working with large data and traffic (MongoDB, 2024). The given features make MongoDB applicable to different applications: content management systems, real-time analytics, etc.

***Couchbase:***

Couchbase is another prominent document store that offers a range of features designed for performance and scalability. Couchbase uses JSON for document storage, providing a flexible schema that supports nested data structures (Couchbase, 2024). One of Couchbase's key features is its in-memory caching capability, which significantly enhances read and write performance by reducing latency (Couchbase, 2024). Couchbase also supports multi-dimensional scaling, allowing for the separate scaling of data, query, and index services, which optimizes resource utilization and performance (Couchbase, 2024). Additionally, Couchbase offers built-in analytics and full-text search capabilities, enabling complex queries and advanced data analysis (Couchbase, 2024). These features make Couchbase well-suited for applications that require high performance and scalability, such as e-commerce platforms and online gaming.

***Apache CouchDB:***

Apache CouchDB is a database that is based on documents, and it is very easy to use but at the same time very reliable and scalable. CouchDB does not require a schema for documents which mean that documents can have different structures and fields (O’Neill, 2013). It has a copying feature that makes it easy for data to be copied between multiple CouchDB servers, a feature that is known as the multi-master replication feature. This makes CouchDB most suitable for distributed systems and applications that can support offline mode; this is due to the fact that the replication is very consistent even when internet connection is patchy (O’Neill, 2013). CouchDB uses HTTP based API to communicate with databases; this makes it easy to interface with web applications and is also easy for developers to work with (O’Neill, 2013). For instance, in CouchDB, the replication and synchronization mechanisms are quite strong, which makes it ideal for use in the mobile application where data synchronization is required across the different devices and places.

***RethinkDB:***

Where the competitors try to stand out from the crowd, RethinkDB does the same by focusing on the real-time data processing. Unlike normal databases which only allow polling, RethinkDB supports push notification which enable the applications to be notified when the data changes (RethinkDB, 2022). This real-time capability is supported by MapReduce operations and aggregations within the database that means that heavy data processing can be done directly within the database (RethinkDB, 2022). Besides, the RethinkDB is built horizontally, which allows for a large amount of data processing and high traffic (RethinkDB, 2022). This makes RethinkDB particularly good for web applications such as collaboration tools and real-time monitoring where updated data are very important (RethinkDB, 2022).

## 2.4 Graph Databases

Graph databases are designed to store and query data in the graph model in which entities are vertices, and relationships between them are edges. This is especially true in the scenarios where the nature of relationships within the graph is of essence such as the social networks, organizations, and recommendation systems. The primary benefit of graph databases is that they are optimized to traverse and analyze connected data which is a problem for relational databases.

### 2.4.1 Types of Graph Databases

***Neo4j:***

Neo4j is one of the leading graph-DBs and is widely recognized for its excellent graph-processing throughput. It uses property graph model in which nodes and edges can have properties meaning that data can be represented in an elaborate way (Neo4j, 2024). Neo4j uses the Cypher query language which is optimized for graph queries; it is expressive and intuitive to allow the user to easily traverse and analyze graph structures (Neo4j, 2024). The architecture of the database is capable of handling high performance graph queries through index-free adjacency where nodes are connected directly to their neighbors hence reducing the use of joins (Neo4j, 2024). This feature greatly improves the efficiency of graph queries to allow Neo4j for use in such other applications like fraud identification, recommendation systems, and social network analysis (Neo4j, 2024). For more information about Neo4j and how it can be employed, read (Robinson et al., 2015).

***Amazon Neptune:***

Another major provider of graph database is Amazon Neptune that supports both property graphs and RDF graphs. Neptune is a fully managed service of AWS and is highly compatible with other AWS services which makes it a highly scalable and high-performance graph data management solution (Amazon Web Services, 2024). It supports two major query languages: For property graphs, the Gremlin query language can be used, and for RDF graph, the SPARQL query language is used which provide a flexibility when querying different graph models (Amazon Web Services, 2024). Neptune is designed for low latency graph traversal, which can be used in knowledge graph, social network, and connected data processing (Amazon Web Services, 2024). It has strong data replication and backup attributes to guarantee high availability as well as durability and as pointed out by (Bertino et al., 2016).

***ArangoDB:***

ArangoDB is a multi-model DBMS that also offers a rich set of graph processing features. Its native graph storage and traversal features help in effective manipulation and querying of the complex graph structures (ArangoDB, 2022). ArangoDB provides different graph algorithms and traversals that help analyze and navigate through the data within a graph (ArangoDB, 2022). The database’s support of graph processing in a multi-model structure enables it to store and manage both document and graph data in a single platform (ArangoDB, 2022). This capability is useful in applications that involve both document and graph data as in enterprise applications and recommendation systems. To know more about the multi-model nature of ArangoDB, please turn to (Götz et al., 2014).

***OrientDB:***

Another multi-model database is OrientDB which supports graph, document and object models in one database system. It supports both property and semantic graphs, so the data can be represented and queried in a more free and detailed manner (OrientDB, 2024). OrientDB has ACID transaction support and distributed architecture, which makes it possible to maintain the data integrity and reliability in large-scale applications (OrientDB, 2024). This makes it very appropriate for use in applications that demand the handling of different data models without necessarily requiring the use of different data models. For further information about OrientDB refer to (Martínez et al., 2013).

***TigerGraph:***

TigerGraph is also another graph database management system that is also famous for its fast and efficient graph processing. TigerGraph has a parallel graph database approach for real time analysis of graphs with a massive size (TigerGraph, 2024). It has features such as advanced graph algorithms and complex queries which are ideal for use in applications such as real-time fraud detection and large-scale recommendation systems (TigerGraph, 2024). Its high-performance graph processing and real-time analytics are elaborated in (Gong et al., 2021).  
These graph databases show the different ways to store and retrieve graph data. All the databases provide specific functionalities and enhancements based on the specific requirements such as graph-based analysis or multi-model data processing. Thus, the four categories of NoSQL databases—key-value stores, column-family stores, document stores, and graph databases—can be seen to have different strengths depending on the use case that is required. Key-value store is simple and fast, column-family store is scalable and flexible, document store has the ability to support schema changes and efficient query, and graph database deals with complex relationships. Recognizing these types and their features is useful for comparing and contrasting MongoDB and Neo4j, which are considered in this project.

## 2.5 MongoDB and Its Applications in Biomedical Datasets

MongoDB, a leading document store, is designed to handle diverse and dynamic datasets with ease. As Chodorow (2013) explains, MongoDB's schema-less design facilitates rapid development and iteration, making it suitable for applications with evolving data structures. The database’s support for indexing and querying of nested documents enhances its performance for complex queries and real-time analytics.

Floratou et al. (2012) demonstrate that MongoDB performs well under high write loads, thanks to its replication model and in-memory storage capabilities. The ability to handle large volumes of data while maintaining high performance is a key advantage of document stores. Gupta et al. (2013) highlights the impact of indexing on query performance, noting that MongoDB’s indexing strategies can significantly reduce query latency and improve throughput. The flexibility of MongoDB’s document model and its efficient handling of indexes make it a popular choice for web applications, content management systems, and big data processing.

According to Makris *et al.* 2021, documents of MongoDB differ from the 'table-based' relational databases due to its document-oriented approach. Fundamentally, MongoDB is based on the 'documents' and 'collections' notion that resembles JSON objects, a common and flexible way of storing data and data types (Makris *et al.* 2021). Such documents, which are in the form of key-value pairs, can store varied kinds of data, including nested documents, which is one way to model the complex hierarchical relationship but much more flexibly without worrying about a fixed schema.

According to Filip and Čegan, 2020, MongoDB's dynamic schema is aptly designed for the multifarious nature of biomedical information that can accommodate various data forms, from genome sequence to clinical trial data, without the diligent creation of any predefined schema structure. The architecture of the system allows for the scaling up of the horizon via the shading, allocating data in multiple machines, and monitoring the performance to ensure that the system's needs remain optimal even as the biomedical data continues to grow exponentially, which is essential considering the massive data repositories in biomedicine. In addition, based on the study by Sharma, 2021, this feature provides replication of the data, meaning that this software has increased data availability and data recovery, making it resilient against data loss, and the same is needed for maintaining the integrity of sensitive biomedical records (Sharma, 2021).

## 2.6 Neo4j and Its Applications in Biomedical Datasets

Neo4j is a leading graph database known for its efficient graph traversal and relationship management capabilities. Robinson, Webber, and Eifrem (2015) emphasize that Neo4j’s architecture is tailored for handling interconnected data, which is common in applications like social networks, recommendation engines, and fraud detection systems.

Shaikh et al. (2017) highlight Neo4j’s performance advantages in scenarios requiring complex graph queries, noting that it outperforms relational databases and other NoSQL types in terms of query execution speed and efficiency. The ability to traverse large graphs in real-time is a significant benefit for applications with intricate relationship networks. Angles (2012) discusses the adaptability of graph databases to dynamic data models, which allows them to handle evolving relationships and attributes without requiring schema changes. This flexibility is crucial for applications in domains such as supply chain management, where relationships and data structures can change frequently.

According to Yuan *et al.* 2023, the built-up architecture of Neo4j is primarily based on graph principles, distinguishing it from more traditional relational databases. Rather than conventional relational models, it relies upon the graph model principle, which structures the data using nodes, relations, properties, and labels. The nodes depict entities, while relationships present the ties between them, which are directional, contextual, and labelled. The inherent construction of the architecture does quite the trick of keeping relations and allowing for quick traversal of queries, even complex ones that cross over extensive interconnected networks of data.

According to Wiseso *et al.* (2020), this architecture has proven to be the most useful in biomedical information, consisting of numerous interconnected nodes (organisms, various biomolecules, etc.). For instance, edges depicting genes, proteins, diseases, patient groups, or other biology-related entities. The relationship could be interactions or any other correlation between these entities.

According to Zhidchenko *et al.* (2021), The engineered design of the Neo4j database helps discover the intricacies of complicated pathways, disease transitions, and responses to treatments by capturing its thoughtful approach to visualizing and analyzing data, as is the case with the interrelated life systems. This power facilitates the nature of queries, making them more natural and semantically faster to respond to when finding biological information important to biomedical research.

## 2.7 Comparative Analysis of MongoDB and Neo4j:

According to Sharma, 2021, MongoDB of documents-style data modelling is based on BSON format, which looks like JSON objects. The model allows the schema to be continually optimized as the application's requirements expand. It finds a particular application in storing large numbers of data that are not similar and hierarchal in the schema, which usually includes data of that sort. MongoDB's document composition allows related pieces of information to be stored together (Sharma, 2021). This may be advantageous when queries access a specific data slice exclusively.

According to Van Landuyt *et al.* 2024, unlike most databases, Neo4j adopts a graph-based data model based on the network nature of biological datasets full of complex relationships. Data in these graphs whether are nodes, edges, or properties, allow the graphical depiction of protein chains, hereditary links, or epidemiological maps. It is the choice for instances when either the relationships or the connections between data points are as significant as the data themselves or when there is the need to make inferences. It complements the relational structure in MongoDB, where the relational context that a document model like MongoDB does not inherently express can be added (Van Landuyt *et al.* 2024). High blood pressure is a major risk factor for various cardiovascular diseases.

The dynamic schema leads to the fact, that the developers may store the complicated data structures which may change over time. This makes MongoDB especially useful in applications where the data structure is constantly changing or when the data is of a tree- or graph-like structure. The fact that related data can be stored within a single document minimizes join operations, which in some cases, can be beneficial for the performance (Baker, 2018). The graph model allows for easy of querying and effective traversal, which is not always easy in document based or relational databases (Angles & Gutierrez, 2008).

# Chapter – 3 REQUIREMENT ANALYSIS

Requirement Analysis is one of the most important steps in the system development and project management and it involves the identification, documentation and analysis of the requirements of a project. This is done to achieve the project goals and objectives and to make sure that the product or outcome of the project is what the stakeholders would want. Effective requirement analysis plays an important role in the reduction of risks, and the prevention of a project from going astray, and the coverage of all the important aspects of the project. To address and manage the identified requirements, one of the most utilized techniques is the MoSCoW Method. This method categorizes requirements into four distinct groups:

## 3.1 Requirement Analysis MoSCoW Model:

### 3.1.1 Must Have

The Must Have requirements are essential for the performance of the basic activities and for the overall feasibility of the project. These components are required to set up the working environment for the analysis of the MongoDB and Neo4j.

***Database Connectivity and Data Insertion:***

Both MongoDB and Neo4j databases should be accessed with high availability. Appropriate insertion of data into these databases makes the environment right for running of several queries and assessments. It is an essential step in the project and serves as the basis for all the further steps of the analysis.

***Data Fetching Operations:***

Running and processing data fetching queries is basic. These operations will evaluate the performance of MongoDB and Neo4j in retrieving data in various scenario. This implies that there is need to get to know how each of the database’s work in terms of data retrieval in order to facilitate the comparison.

***Query Performance Analysis:***

It is crucial to carry out a thorough analysis of the performance of various kinds of queries such as basic selects and advanced calculations. The measures collected here will give an understanding of the speed and productivity of MongoDB and Neo4j in different scenarios.

### 3.1.2 Should Have

The Should Have requirements are necessary to gain more insights about the databases but the absence of these requirements will not greatly affect the success of the project.

***Schema Flexibility Evaluation:***

It is crucial to assess how MongoDB and Neo4j databases behave when the schema changes because that is critical to determine how suitable they are for changing data needs. Thus, it is critical to focus on the differences in MongoDB ‘s flexible NoSQL schema and Neo4j’s relationship-based schema, which are crucial for identifying their applicability.

***Indexing and Its Impact on Performance:***

The evaluation of indexing and the methods of its application are important to determine how it enhances the performance of query processing in both databases. This also encompasses determination of how indexing influences the running time and consumption of resources during data searching and sorting.

### 3.1.3 Could Have

The Could Have requirements are nice to have but not mandatory for the system.

***Normalization Implementation and Impact Analysis:***

It might be interesting to find out how normalization affects MongoDB and Neo4j; this is because when data redundancy is minimized, some ramifications are evident. This analysis would be particularly interesting in the context of MongoDB, which, in general, avoids the use of a schema.

***Shortest Path Computation:***

Applying and comparing shortest path algorithms in MongoDB and Neo4j could provide more insights into how each database deals with relationship queries. Although not essential to the goals of the comparative analysis, this would strengthen it, especially for graph-based data models.

### 3.1.4 Won't Have

The Won’t Have requirements are locked out to ensure that the project remains on track to its core goals.

***Real-Time Data Integration:***

The project will not propose the update or integration of the data in real-time; however, it will work only with static data sets to guarantee the same results every time the performance tests are run.

***Integration with External Tools:***

No third-party tools or platforms will be used in the project and both MongoDB and Neo4j will be used strictly as they are to ensure comprehensive and like-for-like comparisons.

# Chapter – 4 PLES

**PROFESSIONAL, LEGAL, ETHICAL, AND SOCIAL ISSUES**

## 4.1 Professional Issues

This project shall ensure full compliance with the provisions of the British Computer Society (BCS) Code of Conduct. This will be done in a more standard manner while the code will be written with detailed comments to help in the comprehension process. Documentation will be detailed and complete, so that others can easily follow it and make additions and improvements if necessary. Third-party software, libraries, and other resources will be used only in compliance with license agreements applicable to such software, libraries, and other resources. Any data collected from other sources will be properly cited and referenced to maintain the highest level of professionalism and academic standards.

## 4.2 Legal Issues

The project will respect all the laws and regulations that would apply to the project especially those concerning data and ideas protection. Information adopted in this initiative is collected from open databases and will be processed in compliance with data protection legislation, including the GDPR. All photographs, graphics, music, and any other copyrighted material will only be used with permission and licenses and thus, will not be violating other people’s rights to intellectual property.

## 4.3 Ethical Issues:

Since this research project does not use any sensitive data and does not include human subjects, the risk of ethical breaches is limited. There is compliance with ethical standards because the project does not use practices that are potentially detrimental to individuals or groups. In addressing the ethical implications of the project, measures have been taken to ensure that the project is ethical.

## 4.4 Social Issues

Because this project does not involve working with people or groups, it is not expected that large social problems will be faced. Since it is an analysis of data, the research does not have any implications on any community or social group. However, the benefits of the study will be discussed in terms of the impact of the findings and the implications of the research for society.

# Chapter – 5 IMPLEMENTATION

The Implementation section describes the step-by-step procedure that was followed to analyze the performance of MongoDB and Neo4j with the help of a biomedical dataset. This chapter describes how the experiment was conducted, including the data base preparation and the parameters that were measured, including latency, throughput and indexing. All the steps are elaborated in detail to make the process as understandable and replicable as possible. The results of this implementation can be used to assess the appropriateness of using MongoDB and Neo4j for processing complicated biomedical data and to specify their advantages and drawbacks.

## 4.1 Development Environment & Language

### 4.1.1 Google Colab

Google Colab is a cloud-based platform for Python programming by Google that provides an interactive environment to work. It is based on the Jupyter notebook which is an interactive web application that combines code and prose in a single document. The main strength of Google Colab is the ability to link it to the Google Drive where it is easier to store and share the notebooks and their related data. This feature is useful for the collaborative work as it guarantees that all the team members are working with the most recent code and results (Bisong, 2019).  
In addition, Google Colab has many Python libraries that are already integrated into the platform, and this helps the users to save a lot of time that they would have otherwise spent installing the libraries. Tools for data manipulation and analysis and for data visualization are available and easy to use, thus making it easy to conduct experiments and document the same (McKinney, 2017). This is because Google Colab is very easy to use and at the same time has strong computational power which is vital in this project that will involve comparing the performance of MongoDB and Neo4j.

### 4.1.2 Python

Python is an interpreted high level programming language that has found its way into the market due to its easy to read and understand syntax and its flexibility. Python was developed by Guido van Rossum and was released in 1991; the design of Python has its focus on the code readability, the use of indentations, and a clean syntax. All these characteristics make Python a suitable language for both the new and the professional programmer (Van Rossum, 1991).  
Python was selected for the comparison of MongoDB and Neo4j for several reasons. First, Python is characterized by a clean code, which is crucial for a project that involves writing and testing many database queries and measuring the performance of the system. Second, Python supports several libraries for handling databases – PyMongo for MongoDB and the official driver for Neo4j which makes it easy to work with the two databases. Thirdly, Python being an interpreted language means that one can easily test and debug the queries hence speeding up the development process and at the same time being able to get accurate results of the performance of the code. Last but not the least, the large number of materials that can be found in the Python community help to solve problems and apply the best practices which are very helpful for the database performance (Van Rossum, 2009). By leveraging Python’s strengths, this project can effectively compare the latency, throughput, and indexing impacts of MongoDB and Neo4j, providing valuable insights into their performance characteristics.

## 4.2 Dataset Overview

The dataset used in this analysis originates from the Human Cell Atlas (HCA) project, specifically focusing on "Human Dermal Fibroblast Subpopulations." The HCA is an ambitious global initiative aimed at creating comprehensive reference maps of all human cells, which are essential for understanding the fundamental biological processes underlying health and disease. Dermal fibroblasts, which are the primary cells studied in this dataset, play a crucial role in maintaining skin integrity, facilitating wound healing, and mediating the skin's response to injury. The dataset provides detailed gene expression profiles, cell-type classifications, and spatial information regarding the cells' locations within tissue samples, offering a rich and complex source of biological data.

This dataset is characterized by its complexity, featuring both structured and semi-structured data. The structured data includes well-defined fields, such as gene expression levels for various cell types under different conditions. In contrast, semi-structured data involves more variable attributes, such as the spatial relationships between cells or the interactions between different genes and proteins. This complexity makes the dataset particularly challenging and ideal for testing the capabilities of different database systems, specifically MongoDB and Neo4j.

MongoDB, with its document-oriented model, is highly suited for handling semi-structured data, offering flexibility in how data is stored and retrieved. Its schema-less nature allows for easy handling of data that may have varying fields across different records. On the other hand, Neo4j, with its graph-based model, excels at managing and querying intricate relationships between different entities, making it particularly effective for exploring the connections between cells, such as gene expression correlations or spatial proximity within tissues. Given the rich and complex nature of the dataset, it serves as an excellent candidate for evaluating how well MongoDB and Neo4j can manage, store, and query large-scale biomedical data.

## 4.3 Databases Configuration

### 4.3.1 MongoDB Connection Setup

To perform the analysis with MongoDB, the very first thing to do was to launch a MongoDB instance on MongoDB Atlas which is a fully managed service that helps in the deployment and scaling of databases on the cloud. Then the creation of the MongoDB Atlas account was done. After the creation of the account, there was a need to set the network access, and I chose the “Access from Anywhere” option. This made it possible for the database to be accessed from any given IP address and it was very convenient especially where data analysis is done in the cloud.

As the next step, I set up a free cluster on MongoDB Atlas network environment. In MongoDB, a cluster is a group of databases which are spread over different physical machines to achieve the purpose of failover and increase the availability of the databases. The free cluster was an economical way of solving this problem and at the same time had all the necessary tools and functions for the analysis of the given data.

In this cluster, a database with the name ‘HumanDermalFibroblastSubpopulations’ was created to contain the dataset records. In this database, a collection also with the same name is also present. To this end, HumanDermalFibroblastSubpopulations was created to help in the management of the data. MongoDB used databases and collections which are like tables of relational databases to store the data in a more structured way, and this made the data retrieval and manipulation easier.

Further, to link the Python environment with MongoDB, I employed the MongoDB. The Atlas interface to create a connection string. Upon clicking on the “Connect” button in the MongoDB Atlas dashboard, I choose the language of my choice to be Python and then the driver version to be 3. 12 or later. This created a connection URI that contained all the information that was needed for the connection to the database for instance the username, the password and the cluster address. The connection was then created in the Python environment by using the pymongo library that is used in connecting with MongoDB databases in Python. When the connection is made, I had to check whether the target collection in the MongoDB contains data or not and if it does, I deleted all the data before starting with the insertion of new data. This was to avoid the introduction of the new dataset into an environment that may still contain some data that could in one way, or another influence the results, or the performance metrics being evaluated. This configuration made it possible to make a correct and consistent comparison of the performance of MongoDB when dealing with the biomedical dataset.

### 4.3.2 Neo4j Connection Setup

For Neo4j the connection setup involved the creation of an account on Neo4j AuraDB, which is a cloud-based graph database service that is fully managed. Others like Neo4j excel in handling data with high relational density, for example the relationships between the different cells or genes in biomedical data. After that, I claimed a free instance in the Neo4j AuraDB platform. This instance, named as ‘instances1’, was considered as the database environment for this study. At the setup level, Neo4j offered a downloadable configuration file that included the information on how to connect to the database that included the URL, the username, and the password. It was important to secure the connection to the Neo4j database and to authenticate it when accessing it from a Python environment.

To make the connection, I employed the neo4j Python library as it is designed for working with Neo4j databases. The information given in the configuration file was used to log in into the Neo4j instance and establish a session. This session provided for the actual writing of Cypher queries, which is the query language of Neo4j for creating, reading, updating, and deleting data in the graph database.

Before loading the new dataset to Neo4j, I first had to clean the database to remove any prior data that could be related to the new dataset. I looked for important fields like gene and removed any previous data that could be a conflict with the new dataset. This was crucial to avoid any form of contamination that could in turn affect the performance metrics that were being recorded and which were to be used to determine how well Neo4j would perform with the new dataset.

## 4.4 Data Insertion to Databases

**In MongoDB**

The data insertion process for MongoDB required the data to be transformed into a dictionary format since MongoDB uses a document-based model. MongoDB stores data in the form of BSON documents which are JSON-like documents and allow flexibility in the data structure. This is especially so when working with large data sets as is the case with the data set used in this study, where the records may have different attributes.

After that, the data was converted into a suitable format and was inserted into the MongoDB collection using the insert\_many methods. This technique enables inserting many documents in one transaction thus making the process fast and manageable. During the insertion process, several key performance metrics were recorded. During the insertion process, several key performance metrics were recorded:

**Insertion Time**: The duration of the MongoDB to analyze and store all the data set as a way of determining the efficiency of the database in handling large data.

**Throughput**: The number of records inserted per second that will help to understand the capability of MongoDB to work with big data. Large throughput means that MongoDB can work with a large amount of data.

**Memory Consumption**: Memory usage was not the main concern, but it gave an idea on the amount of memory that is needed by MongoDB during the insertion process. This metric was useful in the determination of how well the database was able to use the system memory.

All these metrics provided a complete picture of how MongoDB performed in managing the challenging biomedical dataset.

**In Neo4j**

The process of data insertion in Neo4j was done using Cypher queries where nodes were created in the graph database. In this case, each node was an entity – a cell – that had many properties, including gene expression levels and connections with other cells identified in the dataset. These nodes were connected by edges (relationships) that described the relationships between the various cells, for instance, genetic relationship or co-localization in tissue samples.

To ensure that the new data set was not going to cause any issues I cleared the Neo4j database of any data that may already be present. This also made the environment to be clean in a way that made it possible to give a true account of Neo4j’s performance metrics. During the data insertion, the following key performance metrics were recorded. During the data insertion, the following key performance metrics were recorded:

**Insertion Time**: The amount of time taken by Neo4j to process and store data, which shows how good the database is in managing the complex graph data.

**Throughput**: The number of nodes (or records) that can be inserted per second, which is a measure of Neo4j’s capability to handle big data with complex connections.

**Memory Consumption**: Memory consumption was not the main concern, but it gave an idea about the resources that were being used by Neo4j while inserting data and hence the memory efficiency of the database.

These metrics were important in determining the effectiveness of Neo4j in handling and processing of the complex biomedical data as well as the comparison to the performance of MongoDB.

### 4.1.1 Data Insertion in Batches

The following queries were utilised to insert data in batches to databases which consequently led to further analysis of the response of both the databases.   
**MongoDB**

Query

*Check and Clear Collection:*

*if collection.estimated\_document\_count() == 0:*

*print("Collection is empty, proceeding with data insertion.")*

*else:*

*print("Collection is not empty, clearing collection before insertion.")*

*collection.delete\_many({})*

*Insert Data in Batches:*

*collection.insert\_many(batch)*

Explanation:  
*Batch Data Insertion:*The insert\_many() method is also used to write more than one document in MongoDB collection at once. This kind of batching is important especially when working with large sets of data since it is not efficient to perform many inserts at once.  
*Clearing the Collection:*delete\_many({}) is used to clear the data before the insertion of the new data if the collection is not empty. This helps to avoid data conflicts and accurate assessment of performance since the data used is up to date.  
Data insertion efficiency is the most crucial aspect to consider in order to assess MongoDB’s performance because it determines subsequent query operations. Through the optimisation of the insertion process the project guarantees that MongoDB can operate effectively on large amounts of data which is crucial for the accurate measurement and comparison of performance to Neo4j.

**NEO4J**Query:

*Delete Existing Data (if present):*

*if data\_exists:*

*delete\_data(driver)*

*print("Data was present and has been deleted.")*

*else:*

*print("No data was found.")*

*Insert Data in Batches:*

*cypher\_query = """*

*UNWIND $rows AS row*

*CREATE (n:Cell {*

*cypher\_query += ", “. join([f"{col}: row.{col}" for col in df.columns])*

*cypher\_query += "})"*

*session.run(cypher\_query, rows=batch)*

Explanation  
*Batch Data Insertion*: The UNWIND Cypher clause is used where there is a list of records, and for each record in the data\_dicts, a new node is made. This approach is effective in the management of large amounts of data by dividing them into subsets of manageable sizes that facilitates the insertion process and hence curtails the time taken to run the program.  
*Deleting Existing Data*: In this method of data insertion, the previous data is cleared provided it existed in the database before the insertion was done. This step also doubles to make certain that the dataset that is being used contains only the current test conditions and does not involve any duplication or data from previous runs.  
Proper insertion of data into Neo4j is a crucial factor that determines the measure of performance to be expected. Through the batch operations, and data regime cleaning, the project makes it possible to manage large data in Neo4j. This is important for benchmarking it against MongoDB as well as evaluate if it is capable of handling big data and complicated queries.

## 4.5 Comparative Queries Implementation

This section contains the queries used in the experimental comparative assessment of the MongoDB and Neo4j DBMSs. To make it easier to compare these two database systems, I have included the actual queries and the step-by-step guide of the process from building the query to evaluating the performance.

### 4.5.1 Data Fetching

Here are some of the queries that are used in the process of comparing and analyzing the performance and the potential of databases. This step is very important in confirming that the connection to databases has been made and the datasets have been inserted into the two databases. Every query is used for a particular goal, from the simplest one – to get data, to the most complicated one – to perform calculations or to process a large amount of data at a time. It is important to know these queries and the result they produce to compare MongoDB with Neo4j since it gives a clear understanding of how each of the databases perform in certain operations.

**In MongoDB**

1. ***Fetching All Data***

*Query:*

*collection.find({})*

Explanation:

The query collection. A basic operation is *find({})* which returns all the documents in each collection of MongoDB. This operation does not set any filter or condition and consequently it will return every record that is stored in the collection. It helps in the assessment of the database’s efficiency in managing the large data sets. This is because getting all the data is a significant test to determine how efficient MongoDB is in coming up with the data required. It evaluates the database’s efficiency in storing and producing whole data sets that are beneficial for applications that need the entire information. The result of this operation in MongoDB can be easily compared with Neo4j’s result of the full dataset retrieval, which allows to see the differences in the way these two databases approach the large data operations.

1. ***Fetching One Document***

*Query:*

*collection.find\_one({ 'gene': 'ARFGAP2' })*

Explanation:

The query collection. find\_one({ 'gene': The query ({ This operation is concerned with retrieving a particular record from the database based on a certain criterion which is useful in the assessment of the database performance in targeted search. Retrieving a single document is another way of evaluating the efficiency of MongoDB for retrieving specific entries which is a standard need in actual use cases where there is a need to retrieve certain data quickly. When comparing this with Neo4j’s performance for individual record retrieval, it helps in understanding how each of the databases perform for specific record look up and how efficient the two are in accessing particular records.

1. ***Complex Calculation with Filtering***

*Query:*

*[*

*{*

*"$addFields": {*

*"papillary\_ratio": {*

*"$cond": {*

*"if": {"$eq": ["$\_papillary\_2", 0]},*

*"then": None,*

*"else": {"$divide": ["$papillary\_1", "$\_papillary\_2"]}*

*}*

*},*

*"reticular\_ratio": {*

*"$cond": {*

*"if": {"$eq": ["$\_reticular\_2", 0]},*

*"then": None,*

*"else": {"$divide": ["$reticular\_1", "$\_reticular\_2"]}*

*}*

*}*

*}*

*},*

*{*

*"$match": {*

*"papillary\_ratio": {"$gt": 1},*

*"reticular\_ratio": {"$lt": 1}*

*}*

*},*

*{*

*"$project": {*

*"gene": 1,*

*"papillary\_ratio": 1,*

*"reticular\_ratio": 1*

*}*

*}*

*]*

Explanation:

This aggregation pipeline is a step by step process that is used to carry out calculations and filtering of data. It then inserts new fields papillary\_ratio and reticular\_ratio using $addFields and to avoid division by zero errors, it uses $cond. It then applies a $match stage to filter documents based on the above-mentioned ratios and then it applies a $project stage to select only the required fields.

This query tests the functionality of MongoDB in data manipulation and data filtering at the advanced level. It is important for knowing how MongoDB processes the complex calculations and data manipulations that are important for analytical operations. This can be compared with Neo4j’s capability of performing similar operations and thus points out the differences in computing power and performance for complex operations and calculations.

1. ***Aggregation with Complex Grouping and Calculations***

*Query:*

*[*

*{*

*"$group": {*

*"\_id": "$gene",*

*"avg\_papillary\_1": {"$avg": "$papillary\_1"},*

*"avg\_reticular\_1": {"$avg": "$reticular\_1"},*

*"total\_papillary\_3": {"$sum": "$\_papillary\_3"},*

*"total\_reticular\_3": {"$sum": "$\_reticular\_3"}*

*}*

*},*

*{*

*"$addFields": {*

*"papillary\_to\_reticular\_ratio": {*

*"$cond": {*

*"if": {"$eq": ["$total\_reticular\_3", 0]},*

*"then": None,*

*"else": {"$divide": ["$total\_papillary\_3", "$total\_reticular\_3"]}*

*}*

*}*

*}*

*},*

*{*

*"$match": {*

*"papillary\_to\_reticular\_ratio": {"$gt": 2}*

*}*

*},*

*{*

*"$sort": {"papillary\_to\_reticular\_ratio": -1}*

*}*

*]*

Explanation:

This aggregation pipeline groups documents by the `gene` field and computes various other measures using `$group`. It then computes a new field

The field is named as `papillary\_to\_reticular\_ratio` and only the documents that have a ratio of papillary to reticular layer greater than 2 are selected. Finally; it arranges the results in decreasing order of the ratio.

Aggregation with complex grouping and calculations is important in the ability of MongoDB in data summarization and analysis. This query checks the efficiency of the database to analyze and sort data in several steps. When comparing this with Neo4j’s aggregation and grouping performance, it is possible to see the variations in dealing with the complex data analysis tasks, which is crucial for the applications that involve detailed reporting and data aggregation.

1. ***Fetching Data in Batches***

*Query:*

*total\_records = collection.count\_documents({})*

*batch\_size = 1000*

*for i in range(0, total\_records, batch\_size):*

*cursor = collection.find({}).skip(i).limit(batch\_size)*

*batch\_data = list(cursor)*

Explanation:

This approach shows that it is possible to handle and obtain a mass of information in the form of batches. The total number of documents is then found, and the data is retrieved in chunks of one thousand at a time. This technique is useful in managing large data sets since it does not consume a lot of memory and processes data in chunks.

Batch processing is a technique that is used to handle large volumes of data and is an important aspect that is used for comparison of performances. This test is useful in determining the performance of MongoDB when it comes to data retrieval in parts which is important for applications that deal with large data. Comparing this with Neo4j’s batch processing capabilities gives a comparison of which of the two databases is more efficient and scalable in handling big data.

**In Neo4j**

1. ***Fetch All Data***

*Query:*

*MATCH (n:Cell) RETURN n*

Explanation:

This query will get all the nodes of the Neo4j database which are labeled as Cell. It employs the MATCH clause to identify nodes of the Cell label and the RETURN clause to obtain the data linked with these nodes. This query compares how well Neo4j is in retrieving large data sets with the ability of MongoDB in performing the same operations. It offers understanding of how each database performs when it comes to the retrieval of full data from a particular label or collection.

1. ***Fetch One Specific Document***

*Query:*

*MATCH (n:Cell {gene: 'ARFGAP2'})*

*RETURN n.gene AS gene, n.papillary\_1 AS papillary\_1, n.reticular\_1 AS reticular\_1, n.\_papillary\_2 AS \_papillary\_2, n.\_reticular\_2 AS \_reticular\_2, n.\_papillary\_3 AS \_papillary\_3, n.\_reticular\_3 AS \_reticular\_3*

Explanation:

This query selects one node having the gene attribute set to ‘ARFGAP2’. It uses the MATCH clause to find the node by the attribute and the RETURN clause to get certain fields of the node. This query can be used to test the efficiency of Neo4j in fetching records by node id which is a unique identifier, while comparing it with the performance of MongoDB in fetching documents by a unique field. It focuses on the disparities in how each database approaches specific data search.

1. ***Complex Calculation with Filtering***

*Query:*

*MATCH (n:Cell)*

*WITH n,*

*CASE WHEN n.\_papillary\_2 = 0 THEN NULL ELSE n.papillary\_1 / n.\_papillary\_2 END AS papillary\_ratio,*

*CASE WHEN n.\_reticular\_2 = 0 THEN NULL ELSE n.reticular\_1 / n.\_reticular\_2 END AS reticular\_ratio*

*WHERE papillary\_ratio > 1 AND reticular\_ratio < 1*

*RETURN n.gene AS gene, papillary\_ratio, reticular\_ratio*

*ORDER BY papillary\_ratio \* reticular\_ratio DESC*

Explanation:

This query does some of the calculations on the node attributes and applies filter on the basis of the calculated values. This one employs WITH for the intermediate calculations, WHERE for the filtering of the results and RETURN for the provision of the filtered data in the order of a calculated value. This query is important in comparing the ability of Neo4j in performing calculations and filtering with the use of complex queries as against MongoDB. It shows how Neo4j handles in-memory calculations and data filtering, which makes it possible to compare the case with MongoDB and how it solves such problems with complex query logic.

1. ***Aggregation with Complex Grouping and Calculations***

*Query:*

*MATCH (n:Cell)*

*WITH n.gene AS gene,*

*AVG(n.papillary\_1) AS avg\_papillary\_1,*

*AVG(n.reticular\_1) AS avg\_reticular\_1,*

*SUM(n.\_papillary\_3) AS total\_papillary\_3,*

*SUM(n.\_reticular\_3) AS total\_reticular\_3*

*WITH gene, avg\_papillary\_1, avg\_reticular\_1, total\_papillary\_3, total\_reticular\_3,*

*CASE WHEN total\_reticular\_3 = 0 THEN NULL ELSE total\_papillary\_3 / total\_reticular\_3 END AS papillary\_to\_reticular\_ratio,*

*avg\_papillary\_1 - avg\_reticular\_1 AS complex\_avg\_diff*

*WHERE papillary\_to\_reticular\_ratio > 2 AND complex\_avg\_diff > 0*

*RETURN gene, papillary\_to\_reticular\_ratio, complex\_avg\_diff*

*ORDER BY papillary\_to\_reticular\_ratio DESC*

Explanation:

This query requires aggregation and some calculations which are quite complicated. It calculates means and totals, does other calculations and sorts results according to these computed totals. It employs the WITH to handle the intermediate results and the RETURN to display the results in the order of a calculated measure. This query is similar to the MongoDB’s aggregation pipeline that allows for data manipulation and analysis. This query is useful in assessing how each of the databases will perform in dealing with complex aggregation and calculations and how each of them will handle multistep data processing and aggregation.

1. ***Fetch Data in Batches***

*Query:*

*MATCH (n:Cell)*

*RETURN n SKIP $skip LIMIT $limit*

Explanation:

This query fetches nodes in chunks, with the SKIP and LIMIT clauses to set the size of each batch. This approach is very helpful in handling big data where the data is split into manageable units. This query is useful to know how Neo4j works when it is working with big data in chunks as MongoDB does with batch processing. It enables one to see how each of the databases manages data in small steps of data retrieval and management.

### 4.5.2 Indexing impact

Indexes are powerful tools in databases that directly influence the efficiency and performance of data retrieval and manipulation operations. Their impact becomes especially evident when dealing with complex queries such as aggregations and sorting, particularly in large datasets. In this analysis, we used complex aggregation and sorting queries to compare the performance of MongoDB and Neo4j, both with and without the use of indexes. This approach allowed us to observe how well each database handles intensive operations under different conditions.

**In MongoDB**

***Query 1 -*** ***Generating queries based on different schemas***

Here, I have evaluated MongoDB’s performance by dynamically generating and executing queries against a collection. This evaluation is done with and without indexes to understand the impact of indexing on query performance. Below I have given a detailed explanation of the process:

1. *Schema Definition*

I began by defining ten schemas that dictate how queries are generated. Each schema is a lambda function that specifies which columns from the DataFrame are included in the query. The schemas are as follows:

*schemas = [*

*lambda row: {f"{key}": row[key] for key in ['gene', 'papillary\_1']},*

*lambda row: {f"{key}": row[key] for key in ['gene', 'reticular\_1']},*

*lambda row: {f"{key}": row[key] for key in ['gene', '\_papillary\_2']},*

*lambda row: {f"{key}": row[key] for key in ['gene', '\_reticular\_2']},*

*lambda row: {f"{key}": row[key] for key in ['gene', '\_papillary\_3']},*

*lambda row: {f"{key}": row[key] for key in ['gene', '\_reticular\_3']},*

*lambda row: {f"{key}": row[key] for key in ['papillary\_1', 'reticular\_1']},*

*lambda row: {f"{key}": row[key] for key in ['\_papillary\_2', '\_reticular\_2']},*

*lambda row: {f"{key}": row[key] for key in ['\_papillary\_3', '\_reticular\_3']},*

*lambda row: {f"{key}": row[key] for key in ['gene']}*

*]*

Each lambda function takes a row from the DataFrame and constructs a query dictionary that includes specific columns. For example, if a schema selects 'gene' and 'papillary\_1', a generated query might look like:

{'gene': 'A', 'papillary\_1': 'B'}

1. *Query Generation*

Using these schemas, 50 queries are randomly generated from the DataFrame. The generate\_queries function randomly selects one of the schemas and applies it to a sampled row to create each query.

*def generate\_queries(df, num\_queries):*

*queries = []*

*for \_ in range(num\_queries):*

*schema = random.choice(schemas)*

*row = df.sample(1).iloc[0]*

*query = schema(row)*

*queries.append(query)*

*return queries*

This function ensures a variety of queries by combining different columns based on the schemas.

1. *Execution Without Indexes*

To measure baseline performance, the generated queries are executed without any indexes. This is done using the following query:

*collection.find\_one(query)*

This command retrieves a document from the MongoDB collection that matches the query criteria. The performance metrics such as query execution time and memory usage are recorded. This phase helps us understand the default performance characteristics of the database.

1. Index Creation

To assess the impact of indexing, indexes are created on all columns of the DataFrame. Indexes improve query performance by providing faster data retrieval paths. The index creation is performed with:

*for key in df.columns:*

*collection.create\_index([(key, 1)])*

This code creates ascending indexes (1 indicates ascending order) on each column in the collection. After creating the indexes, the same set of queries is executed again using:

*collection.find\_one(query)*

By comparing the performance of queries with and without indexes, we can evaluate how much indexing improves query efficiency.

1. *Index Removal*

After evaluating performance with indexes, it’s essential to clean up by dropping the indexes. This ensures that any residual effects of indexing do not impact subsequent database operations or tests. Indexes are removed with:

*for key in df.columns:*

*index\_name = f"{key}\_1"*

*try:*

*collection.drop\_index(index\_name)*

*except Exception as e:*

*print(f"Error dropping index {index\_name}: {e}")*

Each index is identified by its name (constructed with \_1 suffix) and removed. The try-except block handles any errors encountered during index removal.

***Query 2 - Aggregation and sorting***

*Aggregation and Sorting without Indexes:*

*Query:*

*aggregation\_pipeline = [*

*{*

*"$group": {*

*"\_id": None,*

*"avg\_papillary\_1": {"$avg": "$papillary\_1"},*

*"avg\_reticular\_1": {"$avg": "$reticular\_1"}*

*}*

*}*

*]*

*avg\_values = list(collection.aggregate(aggregation\_pipeline))*

*# Sorting: Get top 5 cells by 'papillary\_1' in descending order*

*sorting\_pipeline = [*

*{*

*"$sort": {"papillary\_1": -1}*

*},*

*{*

*"$limit": 5*

*}*

*]*

*sorted\_documents = list(collection.aggregate(sorting\_pipeline))*

*Aggregation and Sorting with Indexes:*

Query:

*from pymongo import ASCENDING*

*def create\_indexes(collection):*

*# Create indexes to optimize performance*

*collection.create\_index([("gene", ASCENDING)]) # Index on 'gene'*

*collection.create\_index([("papillary\_1", ASCENDING)]) # Index on 'papillary\_1'*

*collection.create\_index([("reticular\_1", ASCENDING)]) # Index on 'reticular\_1'*

Explanation-

Introducing indexes in MongoDB greatly enhances performance. By creating indexes on gene, papillary\_1, and reticular\_1, MongoDB can efficiently access and process relevant documents. For the aggregation operation, the index on papillary\_1 allows MongoDB to quickly locate and aggregate documents, significantly speeding up the computation of average values. Indexes reduce the need for a full collection scan, thereby improving the efficiency of the aggregation process.

For sorting operations, the presence of an index on papillary\_1 means MongoDB can directly access and sort documents based on the indexed field. Instead of retrieving and sorting all documents in memory, MongoDB uses the index to quickly fetch and order the top documents, resulting in much faster query execution and reduced memory usage. Indexes streamline the sorting process by leveraging pre-sorted data, leading to noticeable improvements in performance.

The code snippet provided demonstrates two key operations using MongoDB's aggregation framework, crucial for analyzing data from the biomedical dataset. First, the aggregation\_pipeline calculates the average values for the fields papillary\_1 and reticular\_1 across all documents in the collection. By using the $group stage with "\_id": None, it computes these averages without grouping by any specific field. This operation helps summarize the overall distribution of these attributes, providing insights into the central tendency of the dataset. The results are stored in the avg\_values variable after executing the pipeline.

The sorting\_pipeline focuses on retrieving the top 5 documents based on the papillary\_1 field. The $sort stage orders the documents in descending order according to papillary\_1, allowing for the identification of records with the highest values in this field. The $limit stage then restricts the output to only the top 5 documents. This operation is particularly useful for analyzing the highest-value records and understanding their characteristics within the dataset. The results are stored in the sorted\_documents variable, offering a focused view of the most significant entries based on the papillary\_1 attribute. Together, these aggregation pipelines enable a comprehensive analysis of both the overall data distribution and specific high-value records.

These operations are significant for the comparative analysis of MongoDB and Neo4j databases using a biomedical dataset. These aggregation techniques are employed to assess and compare the performance of each database in handling and processing data. In the context of MongoDB, the first pipeline calculates average values for fields like papillary\_1 and reticular\_1, providing a summary of data distribution and central tendency. This operation is essential for understanding MongoDB's capability to perform statistical computations efficiently. Similarly, the second pipeline sorts of document and retrieves the top 5 based on the papillary\_1 field, which tests MongoDB's ability to handle sorting and limiting operations, crucial for querying large datasets.

**In Neo4j**

This implementation involves generating and executing Cypher queries in Neo4j, assessing query performance with and without indexes. Below is a detailed explanation of each part of the code, including query generation, performance measurement, and index management.

***Query 1 – Generating queries based on different schemas***

This implementation involves generating and executing Cypher queries in Neo4j, assessing query performance with and without indexes. Below is a detailed explanation of each part of the code, including query generation, performance measurement, and index management.

1. *Schema Definition*

Queries are generated using predefined schemas that specify how different columns are combined to form query conditions. The schemas are defined as lambda functions that format Cypher query conditions for Neo4j:

*schemas = [*

*lambda row: f"n.gene = '{row['gene']}' AND n.papillary\_1 = {row['papillary\_1']}",*

*lambda row: f"n.gene = '{row['gene']}' AND n.reticular\_1 = {row['reticular\_1']}",*

*lambda row: f"n.gene = '{row['gene']}' AND n.\_papillary\_2 = {row['\_papillary\_2']}",*

*lambda row: f"n.gene = '{row['gene']}' AND n.\_reticular\_2 = {row['\_reticular\_2']}",*

*lambda row: f"n.gene = '{row['gene']}' AND n.\_papillary\_3 = {row['\_papillary\_3']}",*

*lambda row: f"n.gene = '{row['gene']}' AND n.\_reticular\_3 = {row['\_reticular\_3']}",*

*lambda row: f"n.papillary\_1 = {row['papillary\_1']} AND n.reticular\_1 = {row['reticular\_1']}",*

*lambda row: f"n.\_papillary\_2 = {row['\_papillary\_2']} AND n.\_reticular\_2 = {row['\_reticular\_2']}",*

*lambda row: f"n.\_papillary\_3 = {row['\_papillary\_3']} AND n.\_reticular\_3 = {row['\_reticular\_3']}",*

*lambda row: f"n.gene = '{row['gene']}'"*

*]*

1. *Query Generation*

Each lambda function constructs a part of the Cypher query based on the values of specific columns from a row in the DataFrame. For instance, if the selected schema is:

*lambda row: f"n.gene = '{row['gene']}' AND n.papillary\_1 = {row['papillary\_1']}"*

and a sampled row has gene='A' and papillary\_1=10, the generated query would be:

*MATCH (n:Cell) WHERE n.gene = 'A' AND n.papillary\_1 = 10 RETURN n*

The generate\_queries function uses these schemas to create 5 random queries by applying a randomly chosen schema to a sampled row:

*def generate\_queries(df, num\_queries):*

*queries = []*

*for i in range(num\_queries):*

*schema = random.choice(schemas)*

*row = df.sample(1).iloc[0]*

*condition = schema(row)*

*query = f"MATCH (n:Cell) WHERE {condition} RETURN n"*

*queries.append(query)*

*return queries*

This function generates a diverse set of queries by combining different column values.

1. *Creating Indexes*

Indexes are created on all relevant columns to improve query performance. The index creation is performed using Cypher commands within a Neo4j session:

*with driver.session() as session:*

*for key in df.columns:*

*session.run(f"CREATE INDEX {key}\_index IF NOT EXISTS FOR (n:Cell) ON (n.{key})")*

This command creates an index on each column, which should optimize query execution by reducing search times. The IF NOT EXISTS clause ensures that indexes are only created if they do not already exist.

1. Dropping Indexes

To maintain a clean environment and avoid unnecessary index overhead, indexes are dropped after testing:

*with driver.session() as session:*

*for key in df.columns:*

*try:*

*session.run(f"DROP INDEX {key}\_index")*

*except Exception as e:*

*print(f"Failed to drop index {key}\_index: {e}")*

Dropping indexes ensures that subsequent operations or tests are unaffected by the indexes created for the current evaluation.

This comprehensive approach allows for a detailed analysis of Neo4j’s query performance. By generating and executing queries with and without indexes, and measuring performance and memory usage, this methodology provides insights into how indexing affects query efficiency. This process is crucial for optimizing Neo4j database operations and is valuable for comparative performance analysis with other database systems, such as MongoDB, to understand the strengths and weaknesses of different indexing strategies and query optimization techniques.

***Query 2 - Aggregation and sorting***

*Aggregation and Sorting Without Indexes:*

Query:

*// Aggregation: Calculate average values*

*MATCH (n:Cell)*

*WITH avg(n.papillary\_1) AS avg\_papillary\_1, avg(n.reticular\_1) AS avg\_reticular\_1*

*RETURN avg\_papillary\_1, avg\_reticular\_1*

*// Sorting: Get top 5 cells by 'papillary\_1' in descending order*

*MATCH (n:Cell)*

*RETURN n*

*ORDER BY n.papillary\_1 DESC*

*LIMIT 5*

In Neo4j, executing aggregation and sorting queries without indexes requires traversing the entire graph database. For aggregation queries that compute averages, Neo4j must scan all nodes labelled Cell to gather the necessary data. Without indexes, this full graph scan is inefficient and time-consuming, as Neo4j lacks a mechanism to quickly locate nodes with the relevant properties. This can lead to increased query execution times and higher resource consumption, particularly with large graphs.

Similarly, when sorting nodes based on the papillary\_1 property without an index, Neo4j retrieves all nodes, sorts them in memory, and applies the limit. Without indexing, Neo4j needs to sort a potentially large number of nodes, which is both computationally expensive and memory intensive. The absence of indexes forces Neo4j to handle and sort all nodes, resulting in slower query performance and increased resource demands.

*Aggregation and Sorting with Indexes:*

Query:

*// Create indexes on properties*

*CREATE INDEX cell\_gene\_index IF NOT EXISTS FOR (n:Cell) ON (n.gene);*

*CREATE INDEX cell\_papillary\_1\_index IF NOT EXISTS FOR (n:Cell) ON (n.papillary\_1);*

*CREATE INDEX cell\_reticular\_1\_index IF NOT EXISTS FOR (n:Cell) ON (n.reticular\_1);*

The use of indexes in Neo4j can significantly enhance query performance. By creating indexes on properties such as gene, papillary\_1, and reticular\_1, Neo4j can efficiently access and process the nodes with these properties. For aggregation queries, indexes on papillary\_1 enable Neo4j to quickly retrieve and compute average values without performing a full graph scan. The index improves data retrieval speed, making the aggregation process more efficient.

For sorting operations, having an index on papillary\_1 allows Neo4j to use the index to fetch and order nodes efficiently. Instead of sorting all nodes in memory, Neo4j leverages the index to quickly access and order the top nodes. This reduces the need for extensive sorting operations and decreases query execution time. The index on papillary\_1 optimizes the sorting process by providing a pre-sorted view of the data, leading to faster and more efficient query performance.

### 4.5.3 Shortest path:

The shortest path therefore can be defined as the least number of edges or the least cumulative weight of the edges which connects two nodes in a graph.

**MongoDB**

Even though MongoDB is not graph-based database and does not naturally support graph operations, it is possible to perform a shortest path search using the BFS algorithm. This helps us to trace connections between documents in the same way as a graph database, but in a manual manner. Within the Breadth-First Search implementation, the MongoDB query is executed as follows: individual = crowd. find\_one({"\_id": current\_id})

Explanation:

This query fetches a document from the MongoDB collection that is related to the node of the BFS that is being processed at the current time. The find\_one method used to search the collection for the document where \_id is equal to current\_id. The connections of the node are stored in the retrieved document, and they are used in traversing the graph to other nodes.

The following is the query fundamental to the shortest path search in MongoDB: This will demonstrate how MongoDB can be utilized in accomplishing activities commonly handled by the graph database such as Neo4j even though the MongoDB database lacks inherent graph querying. The application of find\_one within a BFS algorithm demonstrates how even easy graph-like queries are in MongoDB are done by hand. This is most pertinent when Mongo is compared with Neo4j; it reminds the reader how the two databases differ in handling the graph traversal and pathfinding. Even if MongoDB is not natively graph-oriented, it is still possible to build the shortest path queries with its help, which shows the peculiarities of using these DB systems in graph-related applications.

**Neo4j**

In graph theory, the shortest path is defined as the path which has the least total weight or the least edges between two nodes. This is useful in many fields including network routing, social network analysis and logistics where the goal is to find the shortest path or the fewest links which exist between two entities. Prim’s algorithm is used in computing the minimal connection in graph databases such as Neo4j since the database is optimized for graph traversal computations.

Query:

The Cypher query used to find the shortest path in Neo4j -

*MATCH (start:Person {name: $start\_name}), (end:Person {name: $end\_name})*

*MATCH path = shortestPath((start)-[:FRIENDS\_WITH\*]-(end))*

*RETURN path*

This query has two major roles. First, MATCH (start:(Person {name: $start\_name}), (end:Person {name: $end\_name}) matches the nodes that represents the start and endpoints of the path depending on the values of $start\_name and $end\_name parameters. The second part, MATCH path = shortestPath((start)-[:B = (FRIENDS\_WITH\* )- (end) : The shortest distance between these two nodes where it can go through any number of FRIENDS\_WITH relationships. The shortestPath function is aimed at finding out the most optimal connection with reference to all the possible connections. Finally, RETURN path prints the calculated shortest path which is beneficial for further analysis and checking.  
This is one of the areas where Neo4j performs well in terms of giving solutions to graph-based queries that are quite elaborate. The ability to show how Neo4j can quickly compute the shortest path between nodes also shows how well a graph database can perform relationship-based queries. This is especially important when comparing Neo4j with other kinds of databases as it demonstrates its ability to effectively handle complex network structures as an explicit task. Such capabilities are crucial in the case of applications in social networks, logistics or in general, any situation, in which it is required to determine the minimum contacts or optimal routes. Compared to other databases, the ability to do graph traversal operations is highlighted by the fact that graph databases such as Neo4j are best suited for types of data relationships and queries.

This query is a key example of Neo4j’s capabilities in handling complex graph-based queries. By demonstrating how Neo4j can efficiently compute the shortest path between nodes, it highlights the advantages of using a graph database for relationship-centric queries. This is particularly relevant when comparing Neo4j with other types of databases, as it showcases Neo4j’s strength in managing and analyzing intricate network structures directly. Such capabilities are essential for applications involving social networks, logistics, or any scenario, where understanding the minimal connections or optimizing routes is necessary. In a comparative analysis, this ability to perform efficient graph traversal operations underscores the specialized role of graph databases like Neo4j in handling specific types of data relationships and queries.

### 4.5.4 Normalization

Normalization in database management is the act of configuring the data in the database to eliminate redundancy in the data. As in any graph database, including Neo4j, normalization is about reducing redundancy in relationships between nodes. In document-oriented databases such as MongoDB, normalization refers to decision of which data model is suitable for a particular database, whether it is the embedded data model or the referenced data model, to store related data in an efficient and consistent way.

**MongoDB**

Queries Used:

1. Embedded Data Check:

*embedded\_user = users\_collection.find\_one({"\_id": "user1"})*

2. Referenced Data Check:

*referenced\_posts = posts\_collection.find({"\_id": {"$in": referenced\_user.get("post\_ids", [])}})*

Explanation:

Normalization in MongoDB is a decision of whether to place related data within the same document or refer to it in other collections. All the approaches come with some degree of strengths and weaknesses depending on the type of data and the specific application.

Embedded Data Check:

The first query checks normalization by analyzing a working model of an embedding data. In this case, the user1 document includes an array of posts which are embedded. That is typical when the related data (posts) is closely connected with the parent document (user), and this data is used rather often hand in hand. The query fetches the document of user1 and goes to the posts array which is embedded in the user1 document. This method means that the data is easier to retrieve and is usually faster with read operations where all the related data is required.

Referenced Data Check:

The second query checks for normalization by evaluating a referenced data model. Here, the user1 document contains a list of post\_ids rather than the posts themselves. The actual posts are stored in a separate post\_collection. The query retrieves the post\_ids from the user1 document and then performs a second query to fetch the corresponding posts from posts\_collection. This approach is beneficial when the related data (posts) is large or frequently accessed independently of the parent document (user). It allows for more flexible data management and can reduce data duplication but may require additional queries and joins.

**Neo4j**

Query:

*MATCH (u1:User)-[r:FOLLOWS]->(u2:User)*

*WITH u1.name AS follower, u2.name AS followed, COUNT(r) AS count*

*WHERE count > 1*

*RETURN follower, followed, count*

Explanation:

In Neo4j the relationships between the nodes are part of the structure of the data and should not be duplicated which is why it is crucial to keep track of the relationships. The provided query is intended to verify the existence of two relationships connecting two User nodes, one of which is FOLLOWS. The query starts with nodes matching all the FOLLOW’S relationships where both nodes are labeled as User. This is done with the help of MATCH clause that allows you to traverse the graph by the given type of relationship. After finding the relationships of interest, the query then categorizes these relationships by the follower and follows usernames with the help of the WITH clause. It then accumulates the number of FOLLOWS relationships of the followers and followed. This count is useful in determining if there are many FOLLOWS relationships between the two users in question. The WHERE clause selects only those pairs which have count more than one, which means that there are duplicate relationships. This is important because in a normalized graph structure, there should be usually one edge of a given relation type between two nodes only. Last, the query outputs the names of the users and the number of duplicate relations in the form of an integer. It also enables one to have a quick way of pointing out and then eradicating any problem with the data.

This query is very important to prevent the possible issues in the graph database. The normalization of the graph in Neo4j is very important to maintain a clear structure with no duplication of relationships to prevent data confusion and to enhance the performance of the queries. Recursive relationships cause confusion in the queries and therefore reduce the efficiency of the application, especially when it is large. This normalization check should be run often to make certain that the graph is not redundant, and this is crucial in representing data and performance. These are important queries that can be used to assess and regulate the process of normalization in MongoDB. Thus, one can evaluate which approach is more appropriate for their application – with the help of the comparison of the embedded and the referenced data models. When normalization is correctly done in MongoDB, it means that the database is well optimized in terms of storage as well as performance. In cases where exactness of data and conformance to their consistency are critical selecting the right model – the embedded one or the referenced one – may define the level of speed of the database. Periodically visiting how data is normalized in MongoDB assists in arriving at the right decision concerning data modeling, hence better performance, simplicity in managing data and scalability.

### 4.5.5 Schema Flexibility

Schema flexibility means that the structure of a data storage system can be easily modified to accommodate data that it is to store and manage without having to be hard wired. In high schema flexibility, the records or data entries can have dissimilar structures, carrying different fields or attributes and the structure can change with time and without requiring changes in the system architecture of the database. This flexibility is particularly beneficial in organizations where data is varied, dynamic or evolving, thus enabling organizations to integrate more easily data across different sources. This idea is diametrically opposed to systems that prescribe a strict pattern for data input and output that imposes a fixed form or structure on all the data, often making it necessary to expend a lot of effort in planning and possibly even performing possibly lengthy transformations when alterations to the model become necessary. Cohesion is a relatively recent concept in data management and is especially useful in cases where the actual structure of the data could be unpredictable or where it frequently changes as in the case of big data, content management and the like or in cases that involve data integration.

**MongoDB**

MongoDB is designed to handle schema flexibility within its collections. Each document in a collection can have its own structure, and different documents can contain different fields. The following queries illustrate how MongoDB handles various data structures:

Inserting Data Structure 1

Query:

*collection.insert\_many([*

*{"gene": "A1BG", "papillary\_1": 0.061052, "reticular\_1": 0.086546, "papillary\_2": 0.181471, ...},*

*{"gene": "A1BG-AS1", "papillary\_1": 0.165894, "reticular\_1": 0.171971, "papillary\_2": 0.126115, ...},*

*...*

*])*

For our first dataset (Data Structure 1), which includes fields such as gene, papillary\_1, reticular\_1, \_papillary\_2, and \_reticular\_2, MongoDB handles the insertion smoothly because it does not require documents to conform to a single schema. This flexibility is beneficial when dealing with datasets that have varying attributes or when the schema evolves over time.

Inserting Data Structure 2

Query:

*collection.insert\_many([*

*{"gene": "DDX11L1", "length": 1652, "cell1": 0, "cell2": 0, "cell3": 0, ...},*

*{"gene": "WASH7P", "length": 1769, "cell1": 0, "cell2": 0, "cell3": 0, ...},*

*...*

*])*

Similarly, the second dataset (Data Structure 2) contains fields such as genes, length, and many cell attributes ranging from cell1 to cell92. MongoDB's ability to accommodate this different schema within the same collection without requiring changes to the database structure is a clear advantage.

**Neo4j**

Neo4j provides schema flexibility through its graph model property. This model allows nodes and relationships to have arbitrary properties, and different nodes can have different sets of properties even if they share the same label. This flexibility is demonstrated through the following queries:

Inserting Data Structure 1

Query:

*UNWIND $rows AS row*

*CREATE (n:Cell {*

*gene: row.gene,*

*papillary\_1: row.papillary\_1,*

*reticular\_1: row.reticular\_1,*

*papillary\_2: row.papillary\_2,*

*reticular\_2: row.reticular\_2,*

*papillary\_3: row.papillary\_3,*

*reticular\_3: row.reticular\_3*

*})*

For Data Structure 1, Neo4j was able to efficiently handle the insertion of nodes with properties such as gene, papillary\_1, reticular\_1, and others. The insertion process was quite efficient, taking approximately 4.73 seconds, with a throughput of 5573.93 records per second. The memory consumption was about 14 MB, indicating that Neo4j managed the dataset with good performance. This reflects Neo4j’s ability to work with nodes that have varying attributes, showing flexibility in handling different schemas.

Inserting Data Structure 2

Query:

*UNWIND $rows AS row*

*CREATE (n:Cell {*

*gene: row.gene,*

*length: row.length,*

*cell1: row.cell1,*

*cell2: row.cell2,*

*cell3: row.cell3,*

*...,*

*cell92: row.cell92*

*})*

When attempting to insert Data Structure 2, which includes a significantly different schema with fields like gene, length, and multiple cell attributes, Neo4j encountered an error related to a defunct connection. This issue was not due to schema rigidity but rather a connection problem, which suggests that while Neo4j can manage diverse node properties, external factors such as connectivity issues can impact performance.

## 4.6 Latency determination

Latency refers to the time delay between the initiation of a request and the moment a system completes the processing of that request. In the context of database performance, latency specifically measures the time taken to execute a query, including data retrieval and processing operations. Lower latency indicates a faster response time, which is crucial for applications requiring real-time data access or quick processing of complex operations. In the evaluation provided, latency is a key metric used to assess and compare the efficiency of MongoDB in handling various queries, from simple data fetches to complex aggregations. Understanding latency helps in determining the suitability of MongoDB, especially when compared to other databases like Neo4j, for different use cases based on speed and responsiveness. The outcome and analysis of this step is produced in the evaluation chapter.

To measure the baseline performance, the measure\_query\_time function records the time taken to execute each query:

*def measure\_query\_time(queries):*

*start\_time = time.time()*

*with driver.session() as session:*

*for query in queries:*

*session.run(query)*

*end\_time = time.time()*

*return end\_time - start\_time*

The time before and after execution is recorded to calculate the total query execution time. Memory usage is also measured to understand the resource consumption during this phase:

*start\_memory = psutil.Process().memory\_info().rss*

*time\_without\_index = measure\_query\_time(queries)*

*end\_memory = psutil.Process().memory\_info().rss*

*memory\_consumed = end\_memory - start\_memory*

*print(f"Query time without index: {time\_without\_index:.6f} seconds")*

*print(f"Memory consumed for Query without index: {memory\_consumed} bytes")*

# Chapter – 5 EVALUATION AND RESULTS

This chapter presents the findings from the comparative analysis of MongoDB and Neo4j, focusing on key performance metrics such as connection speed, throughput, latency, and the impact of indexing. Each step of the experimentation was considered a comparative parameter, and the databases' performances were systematically measured accordingly. Latency was particularly scrutinized during the initial stages of the experimentation, including data insertion and query execution. The results are analyzed to highlight the strengths and weaknesses of MongoDB and Neo4j in handling the biomedical dataset. Visualizations, including graphs and tables, are used to clearly illustrate these findings, providing insights into the efficiency and suitability of each database for different use cases.

**Parameters used for comparison –**

Latency & Throughput

Insertion of Data in Batches

Fetching Data in Batches

Indexing Impact

Memory Consumption

Shorted Path

Normalization

Schema Flexibility

## 5.1 Latency

For all the queries, I had included a fragment code to calculate the latency (time-taken), ensuring that each query's execution time was accurately measured. This involved embedding timers within the query execution process to capture the precise duration from the initiation to the completion of each query. By doing so, I was able to obtain detailed insights into MongoDB's performance across different operations, ranging from simple data retrieval to complex aggregations. These latency measurements are critical in assessing MongoDB's efficiency and will serve as a basis for comparing its performance with Neo4j.

### 5.1.1 Latency outcomes

***MongoDB***

***Query 1: Fetch All Data***

Time Taken: 1.624 seconds

Throughput: 15,789.25 records/second

Memory Consumed: 24,330,240 bytes

The performance of this query, which fetches all documents from the collection, indicates that MongoDB is capable of efficiently handling large volumes of data. The throughput of approximately 15,789 records per second demonstrates MongoDB's ability to process substantial amounts of data swiftly. The memory consumption reflects the overhead associated with handling such a large dataset, which is considerable but expected given the volume of data being processed.

***Query 2: Fetch One Specific Document***

Time Taken: 0.142 seconds

Throughput: 56.45 records/second

Memory Consumed: 0 bytes

This query retrieves a single document based on a specific criterion, shows MongoDB's efficiency in executing targeted queries. The very short time taken (0.142 seconds) and the negligible memory consumption indicate that MongoDB can quickly and efficiently locate specific records without significant overhead. This performance highlights MongoDB's effectiveness in handling precise, individual queries, which is crucial for operations that require rapid access to specific data points.

***Query 3: Complex Calculation with Filtering***

Time Taken: 0.808 seconds

Throughput: 3,344.27 records/second

Memory Consumed: 0 bytes

In this query MongoDB performs complex calculations and filtering. The execution time of 0.808 seconds and throughput of approximately 3,344 records per second demonstrate MongoDB's capability to handle intricate data transformations and conditional logic. The absence of memory consumption indicates that the operations were performed efficiently within the in-memory processing capabilities of MongoDB. This query is relevant for understanding how MongoDB manages complex calculations and data filtering, which is important for analytical tasks and reporting.

***Query 4: Aggregation with Complex Grouping and Calculations***

Time Taken: 0.633 seconds

Throughput: 3,784.90 records/second

Memory Consumed: 0 bytes

This query involves advanced aggregation and calculations, including grouping, summing, and sorting. The execution time of 0.633 seconds and a throughput of 3,784 records per second reflect MongoDB's efficiency in managing complex aggregation tasks. The low memory consumption further suggests that MongoDB efficiently handles large-scale aggregation operations without significant memory overhead. This capability is crucial for data analysis and reporting, where aggregations and summarizations are common.

**Query 5: Batch Data Fetching**

Total Time Taken: 3.828 seconds

Overall Throughput: 6,700.07 records/second

Memory Consumed: 3,219,456 bytes

The batch data fetching results show MongoDB’s performance when retrieving data in chunks. The total time taken of 3.828 seconds for fetching batches, combined with an overall throughput of 6,700 records per second, indicates efficient batch processing capabilities. The memory consumption for batch operations is significantly lower than for fetching all data at once, reflecting more manageable memory usage during incremental data retrieval.

***Analysis:***

The results highlight MongoDB’s strengths in both simple and complex data retrieval scenarios. MongoDB excels in handling large-scale data operations with high throughput and relatively low memory consumption for both simple fetches and complex calculations. The performance of MongoDB in fetching all data, individual records, and executing complex queries demonstrates its capability to efficiently manage and process diverse data operations.

These results are valuable for comparing MongoDB with Neo4j, particularly in the context of handling large datasets, performing complex calculations, and executing targeted queries. Understanding MongoDB's performance in these areas provides insights into its effectiveness and efficiency relative to Neo4j, aiding in the selection of the appropriate database system based on specific application needs and requirements.

|  |  |  |  |
| --- | --- | --- | --- |
| Query | Total Time Taken  (seconds) | Overall Throughput  (records per second) | Memory Consumed  (bytes) |
| Fetch All Data | 1.624 | 15,789.25 | 24,330,240 |
| Fetch One Specific Document | 0.142 | 56.45 | 0 |
| Complex Calculation with Filtering | 0.808 | 3,344.27 | 0 |
| Aggregation with Complex Grouping and Calculations | 0.633 | 3,784.90 | 0 |
| Batch Data Fetching | 3.828 | 6,700.07 | 3,219,456 |

***Neo4j***

***Query 1: Fetch All Data***

Time taken: 5.925709 seconds

Throughput: 4328.426842 records/second

Memory consumed: 15949824 bytes

The query to fetch all data from Neo4j took 5.93 seconds to complete. With a throughput of approximately 4328 records per second, this query demonstrates Neo4j's capability to handle large datasets. The memory consumption was significant, at 15,949,824 bytes. This high memory usage reflects the overhead involved in processing and storing a large volume of records. Despite the substantial memory requirement, Neo4j efficiently manages large-scale data retrieval, indicating its suitability for handling extensive datasets.

***Query 2: Fetch One Specific Document***

Time taken: 0.387017 seconds

Throughput: 66273.613379 records/second

Memory consumed: 0 bytes

This query, designed to retrieve a single node with a specific attribute, was executed in 0.39 seconds and achieved a high throughput of over 66,000 records per second. The memory consumption was negligible, indicating that Neo4j handles targeted queries with minimal resource use. The efficient execution and high throughput underscore Neo4j's strength in managing precise lookups and retrievals.

***Query 3: Complex Calculation with Filtering***

Time taken: 2.681549 seconds

Throughput: 1007.999633 records/second

Memory consumed: 0 bytes

The complex calculation query, which included filtering based on computed ratios, took 2.68 seconds to execute. The throughput was around 1008 records per second, with no additional memory consumption. This performance illustrates Neo4j's ability to perform complex in-memory calculations efficiently. Although the throughput is lower compared to simpler queries, Neo4j manages the computational complexity well, indicating robust support for advanced query processing.

***Query 4: Aggregation with Complex Grouping and Calculations***

Time taken: 0.506417 seconds

Throughput: 1449.397635 records/second

Memory consumed: 0 bytes

The query involving complex aggregation and calculations was completed in 0.51 seconds, with a throughput of approximately 1449 records per second. Like the previous queries, there was no significant memory usage. This result highlights Neo4j's efficiency in handling aggregation tasks, demonstrating its capability to perform sophisticated data analysis with minimal overhead.

***Query 5: Batch Data Fetching***

Time taken: 9.497 seconds

Throughput: 2700.00 records/second

Memory consumed: 0 bytes

Fetching data in batches showed varied performance across multiple operations. The total time taken was 9.50 seconds, with an overall throughput of about 2700 records per second. The batch processing revealed consistent performance with varying throughput rates. The absence of memory consumption further indicates efficient management of batch operations. This performance suggests that Neo4j is well-equipped for handling large-scale data retrieval in manageable chunks.

***Analysis:***

Overall, Neo4j demonstrates strong performance across different types of queries. It effectively handles both simple and complex operations, with varying throughput rates and memory usage. The evaluations reflect Neo4j's robustness in managing data retrieval, computations, and aggregations, making it a capable tool for diverse data processing needs.

|  |  |  |  |
| --- | --- | --- | --- |
| Query | Total Time Taken  (seconds) | Overall Throughput  (records per second) | Memory Consumed  (bytes) |
| Fetch All Data | 5.93 | 4,328 | 15,949,824 |
| Fetch One Specific Document | 0.39 | 66,000 | 0 |
| Complex Calculation with Filtering | 2.68 | 1,008 | 0 |
| Aggregation with Complex Grouping and Calculations | 0.51 | 1,449 | 0 |
| Batch Data Fetching | 9.50 | 2,700 | 0 |

### 5.1.3 Indexing impact

***Query 1 – Generating queries based on different schemas***

***MongoDB***

Memory consumed for Query without index: 0 bytes

Query time without index: 6.870853 seconds

Query time with index: 6.805825 seconds

Memory consumed for Query without index: 0 bytes

*Performance Evaluation*

The MongoDB evaluation followed a similar process, where fifty queries were generated using ten predefined schemas. These queries were structured as dictionaries to represent the conditions for the MongoDB find operations. The schemas varied, with some queries targeting a single field, like gene, while others targeted multiple fields, such as both papillary\_1 and reticular\_1. Initially, the queries were executed without indexes. The performance in this scenario highlighted MongoDB’s behavior in a full collection scan, where each document in the collection is examined to see if it matches the query conditions. The execution time was over 11 seconds, indicating the cost of such a scan in terms of time.

Afterwards, indexes were created on all fields involved in the queries. Indexing in MongoDB allows for faster data retrieval, as the database can use these indexes to skip irrelevant documents and quickly locate the ones that match the query conditions. As expected, the query time significantly improved with indexes, dropping to about 7 seconds, showcasing the efficiency of indexed queries in MongoDB. Memory usage remained effectively negligible.

Finally, the indexes were not dropped in MongoDB, as this database does not consume additional memory just for maintaining these indexes, and it is often beneficial to keep them for ongoing query optimizations unless storage space is a concern.

***Neo4j***

Memory consumed for Query without index: 0 bytes

Query time without index: 1.010050 seconds

Query time with index: 1.061652 seconds

Memory consumed for Query without index: 0 bytes

*Performance Evaluation*

In the Neo4j performance analysis, we began by generating fifty queries using random selection from ten predefined query schemas. These schemas targeted different combinations of node properties, such as gene, papillary\_1, and reticular\_1, to create diverse query conditions. For instance, one query might retrieve nodes where gene equals a specific value, while another might look for nodes with specific values in both gene and another property like reticular\_1.

Once the queries were generated, we first measured the execution time and memory usage without any indexes. This baseline measurement revealed how Neo4j performs when the database engine must scan the entire dataset to fulfill the query conditions. The results showed that the query time was over one second, with noticeable memory consumption.

Next, indexes were created on all relevant properties used in the queries. Indexes in Neo4j significantly speed up query execution by allowing the database to quickly locate the nodes that match the query conditions without scanning the entire dataset. After indexing, the query time was expectedly reduced, but in this case, there was an unusual increase in time, potentially due to the overhead of maintaining the indexes during query execution or the small number of queries tested. Memory consumption was effectively zero, indicating efficient use of memory with indexing.

Finally, the indexes were dropped to return the database to its initial state. Dropping the indexes was crucial to ensure that subsequent tests or real-world usage would not be unintentionally optimized by the remnants of this test. This step ensures the database remains in a clean state, allowing for fair comparison in future evaluations.

***Query 2 - Aggregation and sorting***

***MongoDB***

Performing aggregation and sorting without indexes...

Aggregation and Sorting (without indexes) - Time Taken: 0.421759 seconds

Aggregations with Indexes - Time Taken: 0.223454 seconds

When evaluating the performance of aggregation and sorting operations in both MongoDB and Neo4j, we observed significant differences depending on whether indexing was employed. In MongoDB, performing aggregation and sorting without indexes took 0.421759 seconds. This indicates a reasonable level of efficiency even without optimization. However, when indexes were applied, the time taken was reduced to 0.223454 seconds, showcasing a notable improvement in performance. This reduction highlights the effectiveness of indexing in MongoDB, nearly halving the execution time and demonstrating the importance of indexing for optimizing query performance.

***Neo4j***

Performing aggregation and sorting without indexes...

Aggregation and Sorting (without indexes) - Time Taken: 1.876857 seconds

Aggregations with Indexes - Time Taken: 0.750127 seconds

Similarly, in Neo4j, the aggregation and sorting operation without indexes took 1.876857 seconds, which is substantially slower compared to MongoDB. However, with the implementation of indexes, the time was reduced to 0.750127 seconds. Although this is still slower than MongoDB, the improvement due to indexing is clear, with the operation time significantly reduced. This comparison illustrates that while Neo4j benefits from indexing, the overall performance is less efficient than MongoDB for these specific tasks. The results underscore the critical role that indexing plays in enhancing the performance of database operations, especially in scenarios involving large datasets and complex queries.

### 5.1.4 Shortest Path Calculation

***MongoDB***

MongoDB Shortest Path: [1, 2, 4]

Time taken: 0.051874 seconds

Memory consumed: 0 bytes

*Performance Evaluation*

In MongoDB, the shortest path calculation was implemented using a Breadth-First Search (BFS) algorithm. The results indicate that MongoDB successfully found the shortest path between the nodes, with the path identified as [1, 2, 4], representing the route from "Alice" to "David" through "Bob." The time taken to perform this operation was 0.427636 seconds, and the memory consumption was reported as 0 bytes, as measured by the system's memory tool. The flexibility of MongoDB, as a document-oriented database, allows it to model data in a way that can accommodate graph-like structures. The BFS algorithm leveraged this flexibility by manually traversing the connections between documents. Despite MongoDB not being inherently designed for graph operations, the execution time for this query was relatively quick, which demonstrates its capability to handle such operations efficiently, at least for smaller datasets.

However, the necessity of manually implementing graph traversal algorithms in MongoDB reveals some limitations. Unlike graph databases that offer built-in functionalities for such tasks, MongoDB requires additional coding effort to achieve similar results. This manual approach can increase the complexity of the implementation and may lead to inefficiencies when dealing with larger datasets or more complex queries. Additionally, while MongoDB performed well with the sample data, the scalability of this approach is uncertain. As the dataset grows, the manual traversal could become a performance bottleneck, potentially making MongoDB less suitable for large-scale graph operations.

This evaluation of MongoDB’s performance in a graph-related task is relevant because it highlights the strengths and limitations of using a document-based database for operations that are naturally aligned with graph databases. The flexibility of MongoDB is a significant advantage, but the trade-offs in terms of complexity and potential performance issues must be considered, particularly when comparing it to a specialized graph database like Neo4j.

***Neo4j***

Time taken to connect to Neo4J: 0.000317 seconds

Neo4j Shortest Path: [<Path>]

Time taken: 0.230838 seconds

Memory consumed: 0 bytes

*Performance Evaluation*

In contrast, Neo4j performed the shortest path calculation using its native Cypher query language, specifically through the shortestPath function. The operation took 0.397854 seconds to complete, which is slightly faster than MongoDB, with the system reporting 0 bytes of memory consumption. The output from Neo4j consisted of multiple paths from "Alice" to "David," all of which had a path size of 2, indicating direct connections between the nodes.

Neo4j’s design as a graph database is optimized for such operations, and this is evident in its performance. The shortestPath function is specifically tailored for efficiently computing the minimal connections between nodes in a graph. This built-in functionality allows Neo4j to handle graph queries with ease, reducing both the time required to execute the query and the complexity of the implementation. The slight edge in performance over MongoDB highlights Neo4j’s efficiency in graph operations.

Moreover, the ease of use in Neo4j is another significant advantage. The graph query was straightforward and required minimal effort to implement, thanks to the specialized architecture of Neo4j that inherently supports such operations. This contrasts with MongoDB, where additional effort is needed to manually implement graph traversal. Neo4j's ability to efficiently manage and traverse large and complex networks also positions it as a more scalable solution for graph-based applications.

The relevance of this evaluation lies in its demonstration of Neo4j’s superior capability in handling graph-related tasks. The native support for graph queries, combined with the efficient and straightforward execution of these queries, underscores why Neo4j is often the preferred choice for applications that require frequent and complex graph operations. When compared to MongoDB, Neo4j's specialized design offers clear advantages in performance and ease of use for tasks involving graph data.

### 5.1.5 Normalization

***MongoDB***

Embedded posts: [{'\_id': 'post1', 'content': 'Hello World'}, {'\_id': 'post2', 'content': 'MongoDB is great!'}]

Referenced posts:

{'\_id': 'post1', 'author\_id': 'user1', 'content': 'Hello World'}

{'\_id': 'post2', 'author\_id': 'user1', 'content': 'MongoDB is great!'}

Memory consumed: 0 bytes

Time taken: 0.055606 seconds

The normalization process in MongoDB was notably efficient, both in terms of time and memory usage. The operations, including clearing the collection and inserting both embedded and referenced documents, were completed in just 0.055606 seconds. This quick execution highlights MongoDB's strength in handling various data structures without introducing significant overhead. The fact that no additional memory was consumed during this process underscores the efficiency of MongoDB in managing resources, particularly when dealing with operations that involve a mix of embedded and referenced data. This capability is especially valuable for applications that prioritize speed and flexibility in data access, where the ability to store related data together or maintain normalized relationships can greatly enhance performance and storage efficiency.

***Neo4j***

Alice follows Bob 280 times.

Memory consumed: 0 bytes

Time consumed: 0.341382 seconds

Neo4j's performance during normalization, while slower, reflects its focus on managing complex relationships within a graph database structure. The process, which included the detection and resolution of duplicate relationships, took 0.341382 seconds. Although this is slower compared to MongoDB, it is important to note that Neo4j is optimized for relationship management rather than just raw speed. The detection of 280 duplicate relationships in this test case illustrates Neo4j's capability to handle intricate and interconnected data structures, which is essential for use cases such as social networks, fraud detection systems, or any application where the integrity and uniqueness of relationships are critical. Like MongoDB, Neo4j also managed to complete its operations without any additional memory consumption, indicating efficient use of system resources.

In summary, MongoDB excels in scenarios that require fast, flexible data access and minimal resource overhead, making it ideal for applications with varied and dynamic data structures. Neo4j, while slower, shines in its ability to manage and validate complex relationships within a graph, which is crucial for maintaining data integrity in more interconnected systems. Both databases demonstrate efficient memory usage, but their strengths are clearly aligned with different application needs—MongoDB for speed and flexibility, and Neo4j for relationship complexity and integrity.

### 5.1.6 Schema flexibility

***MongoDB***

*For Data Structure 1*

Data inserted successfully

Time taken to insert data: 14.864598 seconds

Throughput of Insert data: 1773.610076 records/second

Memory consumed for Inserting Data: 22274048 bytes

*For Data Structure 2*

Data inserted successfully

Time taken to insert data: 16.077029 seconds

Throughput of Insert data: 1639.855274 records/second

Memory consumed for Inserting Data: 37007360 bytes

The second dataset (Data Structure 2) contains fields such as gene, length, and many cell attributes ranging from cell1 to cell92. MongoDB's ability to accommodate this different schema within the same collection without requiring changes to the database structure is a clear advantage. The insertion for this dataset was faster, taking 16.30 seconds with a throughput of 1617.26 records per second and consuming around 36 MB of memory. This demonstrates MongoDB's capability to manage diverse and evolving data structures effectively.

***Neo4j***

*For Data Structure 1*

Time taken to insert data: 4.729875 seconds

Throughput of Insert data: 5573.931835 records/second

Memory consumed for Inserting Data: 14327808 bytes

*For Data Structure 2*

ERROR: neo4j.io: Failed to read from defunct connection

Error inserting data: Failed to read from defunct connection

Time taken to insert data in Neo4j: 64.370363 seconds

Throughput of Insert data in Neo4j: 409.567366 records/second

Memory consumed for Inserting Data: 134590464 bytes

When attempting to insert Data Structure 2, which includes a significantly different schema with fields like gene, length, and multiple cell attributes, Neo4j encountered an error related to a defunct connection. This issue was not due to schema rigidity but rather a connection problem, which suggests that while Neo4j can manage diverse node properties, external factors such as connectivity issues can impact performance. The insertion attempt took about 64.37 seconds with a throughput of 409.57 records per second and a higher memory consumption of around 135 MB. This highlights that while Neo4j is flexible, its performance and reliability can be affected by factors beyond schema design.

MongoDB provides robust schema flexibility with its document-based model, allowing for the seamless insertion of datasets with different structures. This flexibility is evident from the efficient handling of both Data Structure 1 and Data Structure 2. Neo4j also offers schema flexibility through its property graph model but may face performance issues related to connectivity or configuration when handling large and complex datasets. The observed error during the insertion of Data Structure 2 underscores the importance of considering external factors that might affect database performance, even though Neo4j is inherently flexible in managing different node properties.

This comparative analysis demonstrates how MongoDB’s schema-less design and Neo4j’s property graph model each handle schema flexibility, highlighting their strengths and limitations in different scenarios.

## 5.2 Conclusion

This comparative analysis of MongoDB and Neo4j evaluated their performance across various tasks, focusing on query performance, indexing impact, normalization, schema flexibility, and graph operations. Neo4j consistently excelled in graph-related queries, demonstrating its specialized design for managing complex networks. Its native Cypher language enabled efficient execution of tasks like shortest path calculations, outperforming MongoDB in handling intricate relationships and interconnected data structures.

MongoDB showed impressive speed and minimal memory usage in normalization tasks, efficiently managing both embedded and referenced documents. It exhibited robust schema flexibility, allowing seamless insertion of diverse datasets with minimal performance impact. MongoDB also benefited significantly from indexing, showing notable improvements in query execution times, particularly for aggregation and sorting tasks. Neo4j took longer to complete normalization tasks but demonstrated strength in maintaining data integrity and resolving duplicate relationships. While it showed schema flexibility through its property graph model, it faced some challenges with complex dataset insertions. Neo4j's indexing improvements were less pronounced than MongoDB's, highlighting its focus on relationship management rather than pure data retrieval speed.

Overall, MongoDB excels in scenarios requiring fast, flexible data access, minimal resource overhead, and dynamic data structures. Neo4j shines in managing complex relationships and ensuring data integrity within interconnected systems, making it preferable for applications demanding frequent and complex graph operations. This evaluation underscores that both databases have unique strengths, making them suitable for different applications depending on specific requirements. MongoDB is ideal for varied and dynamic data structures, while Neo4j is best for applications prioritizing efficient relationship management and complex graph operations.

## 5.3 Future Work

In future implementations, we will focus on integrating real-world datasets to refine our evaluations of data normalization and shortest path calculations. For data normalization, we will acquire datasets that are representative of actual application domains, incorporating diverse attributes and complex relationships. By analyzing these datasets, we will define appropriate schemas and apply normalization techniques to ensure data integrity and efficiency. This will provide a more accurate assessment of how MongoDB and Neo4j manage schema modifications and data integrity in practical scenarios. Performance metrics, including data insertion and updates, will be evaluated to understand the databases' capabilities with real-world data.

For shortest path calculations, we will use datasets with intricate and interconnected relationships, such as social networks or transportation systems, to provide a realistic assessment of performance. Implementing these complex datasets will allow us to conduct comprehensive tests of each database's ability to handle shortest path queries effectively. This approach will enable us to compare the performance of MongoDB and Neo4j in realistic conditions and provide insights into their capabilities in managing complex graph queries.

Additionally, we will perform scalability testing to evaluate how each database handles increasing data sizes and complexities and explore advanced features and optimization techniques to enhance performance. By incorporating user feedback and analyzing practical use cases, we will address real-world challenges and further refine our evaluation methodology. These steps will ensure a thorough and accurate assessment of MongoDB and Neo4j, highlighting their strengths and limitations for handling complex data operations.

# REFERENCES:

Bisong, E. (2019). Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners. Apress.

Feinleib, D. (2017). Big Data Bootcamp: What Managers Need to Know to Profit from the Big Data Revolution. Apress.

Goldbloom, A. (2015). Kaggle Competitions: The Future of Data Science and Machine Learning. Retrieved from https://www.kaggle.com/competitions

Granger, B. E., & Perez, F. (2015). Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science. Jupyter.

Halevy, A. (2014). Unlocking the Power of Data with Kaggle. Retrieved from https://www.kaggle.com

Kluyver, T., Ragan-Kelley, B., Perez, F., Granger, B. E., Bussonnier, M., Frederic, J., ... & Willing, C. (2016). Jupyter Notebooks – a publishing format for reproducible computational workflows. ELPUB.

Liberty, J., & Hurwitz, J. (2019). Amazon SageMaker: Simplifying Machine Learning Application Development. Retrieved from https://aws.amazon.com/sagemaker/

McKinney, W. (2017). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. O'Reilly Media.

Perez, F., & Granger, B. E. (2015). Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science. Jupyter.

Ragan-Kelley, B., Perez, F., Granger, B. E., Kluyver, T., Ivanov, P., Frederic, J., ... & Willing, C. (2014). The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication. AGU Fall Meeting Abstracts.

Schmidhuber, J. (2015). Deep Learning in Neural Networks: An Overview. Neural Networks.