The University of Western Ontario
Department of Computer Science

Neo4J and Cypher

Group Members:
Samuel Catania
Jonathan Lee
Smitkumar Patel
Reese Collins
Brendan Bain

Databases II (Advanced Databases)

Project Report

December 24, 2023

Contents

1 Architectural Analysis **3**

1.1 The Graph Storage and Schema-Optional Model 3

1.2 Cypher Query Language and Security 3

1.3 Design Principles 3

1.4 ACID Compliance and Transaction Management 4

2 Functionalities and Features **4**

2.1 Neo4J 4

2.1.1 Security 4

2.1.2 Data Storage 5

2.1.3 Query Optimization 5

2.1.4 Comparison 6

2.2 Cypher Query Language 6

2.2.1 Overview 6

2.2.2 Query Semantics 6

2.2.3 Syntax 7

2.2.4 Strengths and Weaknesses 8

3 Case Study **10**

3.1 Overview 10

3.2 Knowledge Graph 10

3.3 Scenario 11

3.4 Result and Conclusion 11

4 Performance Tests **12**

4.1 Overview 12

4.1.1 The Stack Overflow Database Schema 12

4.1.2 Measurement Approaches 13

4.1.3 Performance and Scalability Tests 13

4.1.4 Purpose 13

4.2 Index-Based Query Performance 13

4.2.1 Overview 13

4.2.2 Methodology and Execution 13

4.2.3 Performance Evaluation Queries 14

4.2.4 Results and Observations 14
4.3 Graph Algorithm Efficiency Test 15
4.3.1 Overview 15
4.3.2 Methodology and Execution 15
4.3.3 Results and Observations 15
4.4 Virtual Graph Creation Test 16
4.4.1 Overview 16
4.4.2 Methodology and Execution 16
4.4.3 Results and Observations 16
4.5 Data Scalability Test 17
4.5.1 Overview 17
4.5.2 Methodology and Execution 17
4.5.3 Results and Observations 17

5 IGNORE BELOW 18

1 Architectural Analysis

Neo4j is a high-performance, NoSQL graph database that utilizes Cypher Query Language and supports the full range of Create, Read, Update and Delete (CRUD) operations on graph data models. Neo4j offers 3 distinct advantages due to its graph model structure:

1.1 The Graph Storage and Schema-Optional Model

Neo4j's storage engine stores nodes, relationships, and properties in a graph-model-optimized manner - making it ideal for graph operations. One of the main advantages of this database over relational databases is it's design for effective relationship traversal. The architecture that makes up Neo4j ensures consistency and dependability across remote contexts. It does this by managing data replication using a leader-follower mechanism. Additionally, Neo4j uses causal clustering for robustness and scalability. By using this method, clusters are guaranteed to be reliable and able to manage high data and query volumes.

Altogether, Neo4j provides freedom in developing data models by enabling execution without a set schema. As such, Neo4j is schema-optional, where it enables the creation and use of indexes to not only improve efficiency, but to enforce restrictions on functions like existence and uniqueness within data. This ability to function without a set schema provides a fantastic amount flexibility for data models that are always growing and changing.

1.2 Cypher Query Language and Security

Cypher is a declarative graph query language that allows for the efficient querying of graph databases. By using an SQL-like syntax that makes it very easy to read and write, Cypher increases the ease in which the data in a graph database can be manipulated. Furthermore, its ability to perform complex pattern matching, a mechanism used to navigate, describe and extract data from a graph by applying declarative patterns, unlocks even more potential for the users of graph databases like Neo4j.

Additionally, Neo4j supports enterprise-level security requirements with features including encrypted connections, LDAP integration, and role-based access management. Neo4j provides thorough audit logging for compliance, tracing who accessed or changed data and whenever. Neo4j users can add custom methods written in Java or other JVM languages to expand its capability, too! Even more so, Neo4j integrates effortlessly into a variety of development environments and has official drivers for languages including Java, Python,.NET, JavaScript, and more.

1.3 Design Principles

Unlike relational databases that structure data in tables, Neo4j represents data with nodes and relationships. Each node represents an entity (like a person, place or thing), and each relationship represents how two nodes are associated. Furthermore, unlike other databases, relationships take first priority in graph databases - resulting in a data model that is both significantly simpler and more expressive than other relational or NoSQL databases. This provides useful key advan- tages, like how graph databases don't have to infer connections using things like foreign keys or out-of-band processing.

Another key feature Neo4j provides is the ability to place attributes (key-value pairs) onto these node-and-edge relationships. These attributes (or labels) can be placed on both nodes and the types of relationships between nodes, and overall allow for a more effective data structure when understanding and trying to query the database. Neo4j indexes using these labels, which expedites the retrieval of data.

Finally, Neo4j's query language, Cypher, is excellent at expressing complicated relationships and patterns inside the graph. For ease of understanding and clarity, it employs ASCII-Art syntax. Like SQL, Cypher supports ordering and aggregating functions, which makes it flexible for a range of query needs.

1.4 ACID Compliance and Transaction Management

Data consistency is essential, particularly in complex processes involving numerous graph members. Neo4j ensures that every transaction complies with ACID. Neo4j effectively manages several concurrent transactions by locking resources at the lowest feasible level (relationships and nodes) in order to increase throughput. Every node in Neo4j makes a direct reference to the items next to it. This in turn results in faster graph traversals result from this design's removal of the requirement for index look-ups when traversing relationships.

2 Functionalities and Features

2.1 Neo4J

2.1.1 Security

Neo4j employs schema based security, controlling users' ability to access and read different parts of the graph within the database. This ensures that only authorized users have access to only the data that they need, reducing the chances for breaches to occur within the database.

Furthermore, Neo4j creates security event logs and query logs to improve the overall security of the database. They can be helpful for monitoring and analyzing what users are trying to do when accessing the database, which helps keep an eye for any potential breaches. The data stored is all encrypted using intra-cluster encryption, further increasing the safety of the data. Additionally, to keep to industry standards, Neo4j is also SOC 2 Type II compliant. This compliance can only be obtained through an outside auditing procedure and demonstrates a given service provider can securely manage its data to protect the interests of its organization and the privacy of its clients.

2.1.2 Data Storage

As a graph based database, Neo4j stores data within nodes. As such, the structure of files is different than a traditional database, as the following four data types must be stored: **nodestore***, **relationship***, **property*** and **label***. Each file stores these nodes, relationships created and defined by graphs, key/values of the database and index related label data created from graphs respectively. Neo4j uses a schema-less database, by using fixed record lengths and offsets to store data and find them for queries. The data/nodes within graphs are stored within a linked list of fixed size records, with each having a key and value pointing to the next property. Furthermore, each node points to the start and end nodes as well as previous and next nodes.

2.1.3 Query Optimization

While there are multiple ways queries can be optimized in Neo4j databases, indexes or planner hints are the main two methods.

Indexes

Indexes are used to solve the combination of relationship and property of a node, it is most often used for queries. There are different types of indexes in Neo4j databases used to solve different searches optimally. The indexes are **LOOKUP**, **RANGE**, **POINT** and **TEXT**. **LOOKUP** index solves for node label and relationship conditions. **RANGE** is more versatile and supports most types of queries including node and relationship conditions searched by **LOOKUP** index. The **POINT** index solves for conditions that operate on points, making it only usable when condition points to null for all non-point values within the query. These point values are points within a two dimension or three dimensional plane of an object. Lastly the **TEXT** index only operates on conditions that search for strings. For example, the **TEXT** index works on conditions like `n.name ↪ = "Joe"` or using keywords in the condition such as **STARTS WITH**.

Planner Hints

Planner hints are also helpful with optimizing queries as it can be used to influence decisions of the planner when building an execution plan for a query. This is done through the **USING** keyword within the query. When there are multiple possible starting points for a query, the planner may choose a sub-optimal starting point. Through the **USING** keyword, this can be influenced in three main ways, index hint, scan hint and join hint. Index hint specifies which index to use as a starting point for the query. Scan hint can be used when a query matches a large portion of an index, making a scan to filter out rows and columns potentially faster to proceed with. A join hint is used to enforce certain joins within specific points of a query.

2.1.4 Comparison

Comparing Neo4j to SQL, a relational database management system, Neo4j's data units are nodes and relationships, both of which can have their own properties which correspond to attributes in RDBMS. For relationships between two entities, there is no need to use foreign keys like in RDBMS, as relationships have full references without any constraints as it is not optional in Neo4j. Due to data in graph structures within Neo4j databases, it is suitable for situations where relationships between nodes are as important as the entities themselves. It excels in querying for complex relationships, graph-related queries such as searching for paths and analyzing connected data to uncover patterns. As compared to SQL which is efficient for structured data, SQL may face performance issues when dealing with issues where Neo4j excels at.

2.2 Cypher Query Language

2.2.1 Overview

As a declarative graph querying language meant for a wide range of end users, Cyphers syntax was designed around data relationships and human readability. The visual approach taken is meant to resemble the elements of the graph being queried, through nodes and relationships in the form of () and $\rightarrow$ respectively. Cyphers unique syntax creates a layer of abstraction and puts emphasis on what is being returned out of the graph, rather than the process in how that information is fetched. This makes Cypher a very easy language to learn as its highly intuitive.

2.2.2 Query Semantics

Cypher queries are based on node and relationship patterns. The following query uses only nodes to match people named Bob, and returns their birthdays:

```
MATCH (n:Person {name: 'Bob'})
RETURN n.birthday AS Birthday
```

Most Cypher queries will make use of the **MATCH** and **RETURN** clauses as they are fundamentally the select and project operators in relational algebra, respectively. A node pattern happens within the (). The example above binds the results of the query to the variable *n*. The colon after the variable indicates that a node should be considered in the match if it has the label "Person". Labels are used to give nodes a certain classification, and a single node can have multiple labels at a time. `"{name: 'Bob'}"` is an example of a property key-value expression, and indicates the property of the Person nodes "name" must be "Bob". The **RETURN** clause is asking for the variable to output the property "birthday" from the nodes gathered as n, and to display them named "Birthday" through the **AS** clause. The following query uses nodes and relationships to return the number of friends Bob has had for at least 3 years:

```
MATCH (:Person {name: 'Bob'}) -[friends:FRIENDS_WITH
WHERE friends.since < 2021]->(friend:Person)
RETURN count(friends) AS numberOfFriends
```

A relationship is denoted by `"-->"` to show the direction of the relationship, and must have a starting node and end node on both sides of the arrow. Specifics of a relationship to be queried are put in square brackets to form the relationship pattern ()-[]->(). Like nodes, relationships can be assigned to a variable, like in this case with the name "friends". Relationships also have properties that can be used in the query, which can be seen in the **WHERE** clause. The **WHERE** clause is used within nodes and relationships to set conditions on their properties. In the example query, the condition is set on the **FRIENDS_WITH** relationship property ".since" to be smaller than 2021. The **RETURN** clause makes use of the **count()** function, which returns the number of instances in a variable.

2.2.3 Syntax

Naming Rules and Conventions

Like all languages, Cypher has a set of naming rules that apply to variables, indexes, node labels, relationship types, property names, and constraints. Names are case-sensitive, meaning "name", "Name", and "NAME" all refer to different names. Names may contain non-english characters and numbers, but cannot start with a number (E.g. "1name"). The longest name allowable 65535 characters, and whitespace characters are removed automatically. Names cannot contain any symbol other than '$' and the underscore. If '$' is to be used, it must go at the beginning of the name to indicate it is a parameter. The number, symbol, and whitespace rules can be bypassed with the use of backticks to create an escaped name (E.g. ‘1valid‘, ‘also%valid‘, ‘this is valid‘ are all valid names). For easier distinguishability between node labels and relationship types, node labels use Camel-case starting with an uppercase as a naming convention (e.g. `:MusicianName`), and relationship types use uppercase with words separated with underscores (e.g. `:PLAYS_INSTRUMENT`).

Syntax of Node Patterns

```
nodePattern ::= "(" [ nodeVariable ] [ labelExpression ] [
    ↪ propertyKeyValueExpression ] [ "WHERE" booleanExpression ] "
    ↪ )"
```

Syntax of Relationship Patterns

```
relationshipPattern ::= fullPattern | abbreviatedRelationship
```

```
fullPattern ::= "<-[" patternFiller "]-" | "-[" patternFiller "]->
    ↪ " | "-[" patternFiller "]-"
```

```
abbreviatedRelationship ::= "<--" | "--" | "-->"
```

```
patternFiller ::= [ relationshipVariable ] [ typeExpression ] [
    ↪ propertyKeyValueExpression ] [ "WHERE" booleanExpression ]
```

Operators

The following is a table of operations available in Cypher from the official Cypher documentation:

Operator Category	Operators	Description
Aggregation operators	DISTINCT	Used to remove duplicate values in RETURN clause
Property operators	., [], =, +=	. is used for static property access, [] is for dynamic property access, = for replacing all properties, += for mutating specific properties
Mathematical operators	+, -, *, /, %, ^	Self-explanatory
Comparison operators	=, <>, <, >, <=, >=, IS NULL, IS NOT NULL	Self-explanatory
String comparison operators	STARTS WITH, ENDS WITH, CONTAINS, =~	=~ is used for regular expression matching
Boolean operators	AND, OR, XOR, NOT	Self-explanatory
String operators	+	Used to concatenate strings
List operators	+, IN, []	+ is used for list concatenation, IN is used to check if an element exists within a list, [] is used for accessing elements dynamically

2.2.4 Strengths and Weaknesses

Strengths

- **Graph-Focused:** Cypher is specifically designed for querying graph databases, making it highly effective for tasks related to graph traversal, pattern matching, and graph analytics as it is optimized for expressing relationships and patterns in the data. Some examples where Cypher would excel include supply chain management databases, healthcare databases, and network infrastructure databases.

- **Pattern Matching:** Cypher excels at pattern matching, allowing users to express complex graph patterns with ease, which is helpful for uncovering relationships and dependencies within the data.
- **Declarative Nature:** Cypher queries are declarative, meaning that users specify what they want to retrieve rather than detailing how the database should retrieve it. This simplifies the query-writing process and allows for more intuitive queries.
- **Readability:** Cypher queries are often more readable than equivalent queries in other languages, especially when dealing with complex graph patterns. This makes it easier for developers and analysts to collaborate and understand each other's queries.
- **Support for Property Graph Model:** Cypher is designed to work seamlessly with the property graph model, which is a flexible way of representing and storing graph data. This model includes nodes, relationships, and key-value properties, allowing for richer and more detailed data representation.

Weaknesses

- **Limited to Graph Databases:** While Cypher is excellent for graph databases like Neo4j, it is not suitable for relational databases or other types of data stores. This limits its applicability in environments where multiple types of databases are used.
- **Less Mature Ecosystem:** Cypher may have a less mature ecosystem compared to more established query languages like SQL. SQL has a broader range of tools, libraries, and community support due to its longer history and widespread use.
- **Learning Curve:** While Cypher is designed to be user-friendly, there is still a learning curve, especially for those who are new to graph databases. Users familiar with SQL might find it takes some time to adapt to the graph-oriented thinking required by Cypher.
- **Syntax:** Although the syntax is also a strength of Cypher, having to type out the visual elements of nodes and relationship can be slow to type and may feel redundant.
- **Performance Concerns:** While Cypher is generally performant for typical graph queries, there can be performance concerns with very large datasets or complex queries. Optimization may require a deeper understanding of the underlying graph database engine.

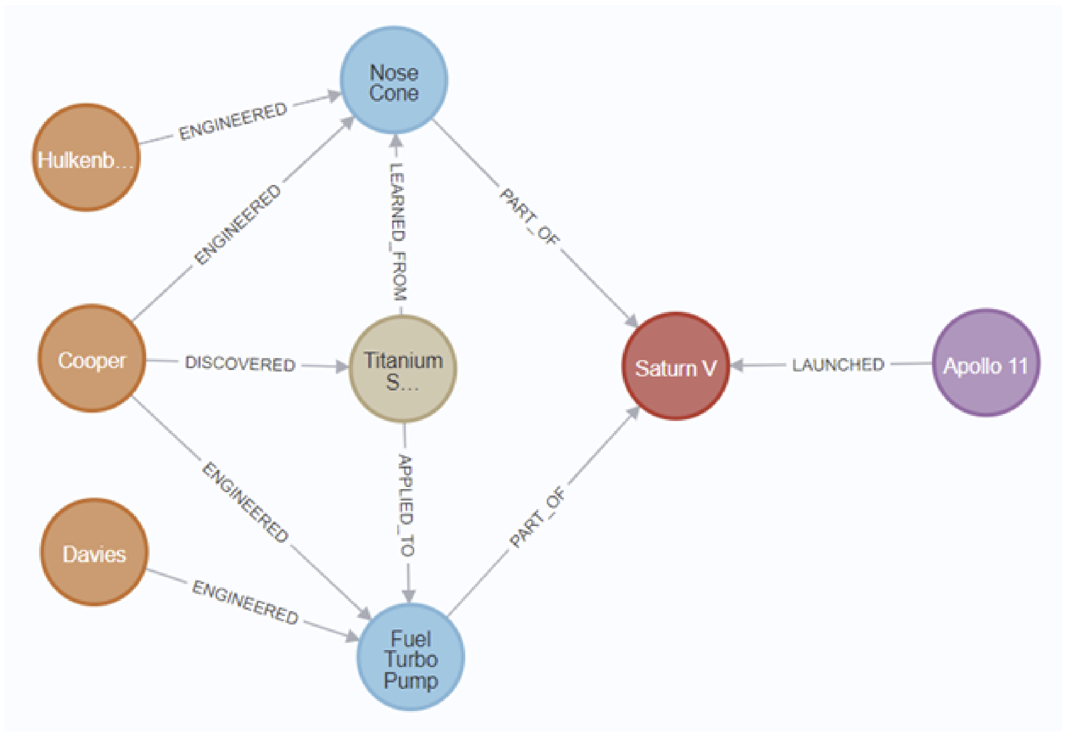


Figure 1: The contents of the NASA knowledge graph.

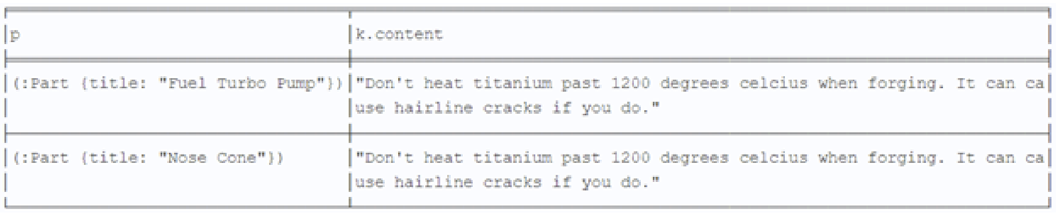
3 Case Study

3.1 Overview

As a graph database, Neo4j excels in situations with highly relational data. It is extremely quick at performing what would be joins in a relational database, especially when you start chaining joins together. It is for this reason that it is often used in the context of "knowledge graphs." NASA is just one organization who has leveraged the power of Neo4j as a knowledge graph (Meza, 2021). By mapping out their previous missions and the lessons learned from them, NASA employees were able to avoid the "siloing" problem a lot of organizations that size face. Engineers could search the database to view any of NASA's previous missions, the lessons they learned from the mission and how those lessons were used in other missions. It was all tied together in such a way that engineers didn't have to spend valuable time and resources re-learning something NASA had already done in a different department.

3.2 Knowledge Graph

We can explore this functionality by creating our own knowledge graph. We will replicate the use case at NASA: mapping projects and the knowledge gained. For this example, I have kept the scope small. One could imagine the scope of the actual knowledge graph at NASA is quite massive. Below, Figure 1 contains the contents of the graph. Here we have five categories of nodes: People, Parts, Rockets, Knowledge, and Missions.



```
|p                               |k.content                                                  |
|(:Part {title: "Fuel Turbo Pump"})|"Don't heat titanium past 1200 degrees celcius when forging. It can ca|
|                                |use hairline cracks if you do."                              |
|(:Part {title: "Nose Cone"})    |"Don't heat titanium past 1200 degrees celcius when forging. It can ca|
|                                |use hairline cracks if you do."                              |
```

Figure 2: The results of running the query.

3.3 Scenario

Let's say you were an engineer who was about to create a brand-new part. You know that the part needs to be made of titanium to resist the stresses and meet weight requirements. In a production environment, you would go to the user-interface that has been built to search the database, but for this example, we will be running Cypher queries manually. You want to know all knowledge relating to titanium as well as what projects the person who discovered the knowledge worked on. For this, the following query would be used:

```
MATCH (k:Knowledge)<-[:DISCOVERED]-(m:Person)-[:ENGINEERED]->(p:Part) WHERE k.title CONTAINS "Titanium" RETURN p, k.content
```

3.4 Result and Conclusion

After running this query, you are provided with the following results as seen in Figure 2:

With this information, time and money is saved as you don't have to repeat the discovery that your colleagues have already made. This is why knowledge graphs are an extremely popular application of Neo4j. Organizations like NASA are so huge that there is no possible way for a single person to know everything that is going on within the organization. By creating a central database of past, present, and future projects, it gives a better overview of the work being done in the organization and can reduce redundancy.

This was a very brief view at the true power of Neo4j. To explore every feature would take far too long, as the DBMS has been in constant development since 2007. However, being a DBMS, it comes with many of the features that are already well understood such as full-text search, subroutines, etc. Plus, with a host of plugins that can extend the functionality, if you can dream it, chances are there is a plugin that can do it.

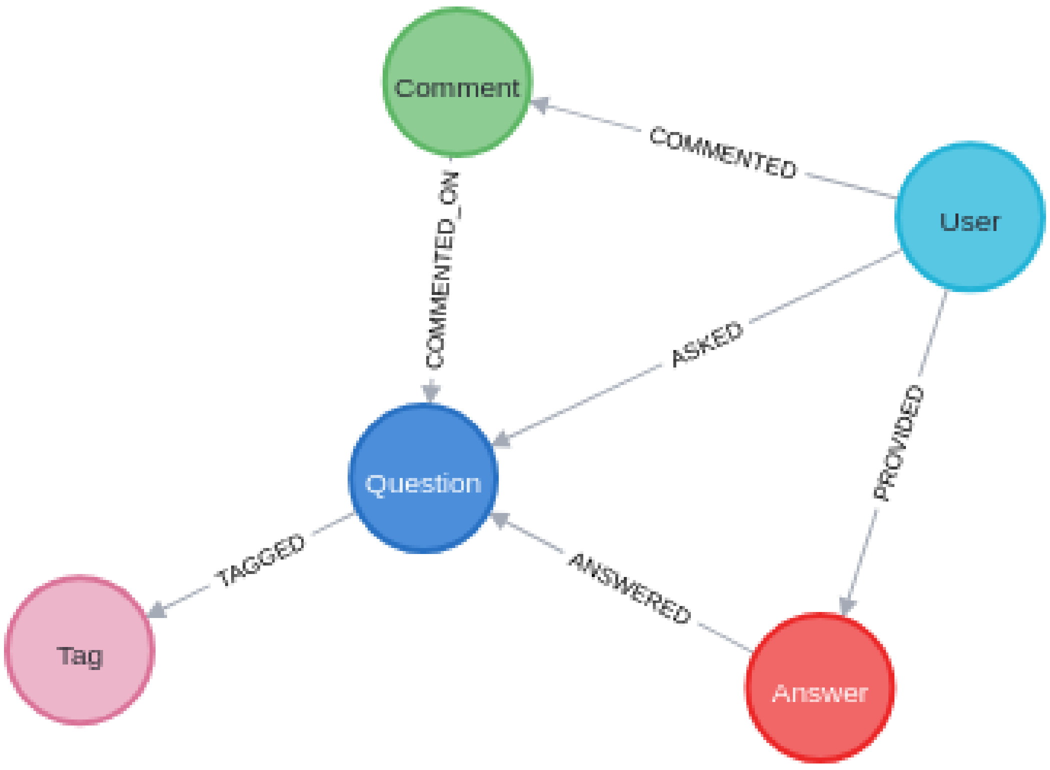


Figure 3: The schema that describes the Stack Overflow database.

4 Performance Tests

4.1 Overview

As with any database, its important to run a multitude of tests on various aspects and features of Neo4j in order to obtain measures of actual performance and scalability, not just theoretical. To do so, a database full of data obtained from Stack Overflow can be utilized. Stack Overflow, known for its extensive web of user interactions, questions and zanswers regarding topics of computer science and technology, presents an ideal case study for assessing the capabilities of Neo4j.

4.1.1 The Stack Overflow Database Schema

The Stack Overflow dataset comprises various node types, including Users, Questions, Answers, Comments, and Tags. This diversity allows for a comprehensive evaluation of Neo4j's ability to handle complex, real-world data structures. The dataset features intricate relationships like users posting questions (**ASKED**), commenting on questions (**COMMENTED_ON**), and providing answers (**PROVIDED**). Additionally, Stack Overflow's continuously growing database offers a practical scenario to test scalability. It reflects common challenges faced in managing and querying large datasets. Figure 3 provides a visualization of the Stack Overflow database schema:

4.1.2 Measurement Approaches

Execution time will be used as the primary metric for assessing query performance, particularly due to the lack of direct resource utilization metrics in the online sandbox environment. Additionally, Query Plan Analysis will be conducted with the **PROFILE** command to gain insights into query execution strategies and index efficiency.

4.1.3 Performance and Scalability Tests

The following performance and scalability tests will be conducted:

- **Index-Based Query Performance:** Evaluates how indexes affect query execution times.
- **Graph Algorithm Efficiency:** Assesses the performance of graph algorithms on large datasets.
- **Virtual Graph Creation:** Examines the impact of creating virtual graphs on system performance.
- **Data Scalability:** Tests how increasing data volumes impact query execution and resource usage.

4.1.4 Purpose

The goal of this exploration is to understand how Neo4j handles a dataset as complex and extensive as Stack Overflow's. By analyzing various performance and scalability aspects, valuable discoveries can be made when it comes to optimizing and deploying Neo4j in scenarios that demand the handling of large-scale, intricate graph data structures.

4.2 Index-Based Query Performance

4.2.1 Overview

The efficient management of graph databases like Neo4j depends on the effective use of indexes. This test attempts to evaluate the creation and use of indexes in Neo4j on query performance.

4.2.2 Methodology and Execution

The test focused on assessing the impact of indexes on the retrieval speed and resource usage for specific queries. The initial step involved creating indexes on various node properties that targeted properties frequently accessed in queries. The following Cypher commands were executed to create these indexes:

```
CREATE INDEX ON :User(uuid);
CREATE INDEX ON :Question(uuid, creation_date);
CREATE INDEX ON :Answer(uuid, score);
CREATE INDEX ON :Comment(uuid, score);
CREATE INDEX ON :Tag(name);
```

4.2.3 Performance Evaluation Queries

After establishing the indexes, specific queries were executed to evaluate the impact of these indexes on performance. These queries included:

Most Active Users Query

This query aimed to identify the top 10 users with the highest contributions (questions and answers combined).

```
MATCH (u:User)-[:ASKED|:PROVIDED]->(p)
RETURN u.uuid, COUNT(p) AS contributions
ORDER BY contributions DESC
LIMIT 10;
```

Popular Tags Query

This query aimed to identify the top 10 most popular tags based on their association with questions.

```
MATCH (t:Tag)<-[:TAGGED]-(q:Question)
RETURN t.name, COUNT(q) AS question_count
ORDER BY question_count DESC
LIMIT 10;
```

4.2.4 Results and Observations

The use of **PROFILE** with these queries revealed significant reductions in execution times post-indexing, particularly for queries involving multiple joins. The 'Most Active Users' query, for instance, showed a marked decrease in database hits, indicating more efficient data retrieval due to index utilization. The execution plans shifted from full node scans to targeted index lookups, demonstrating the effectiveness of the indexes.

This test conclusively demonstrated the substantial impact of index creation in enhancing query performance within Neo4j. By facilitating quicker access to data, especially in complex queries, indexes emerged as essential tools for optimizing database operations in Neo4j environments.

4.3 Graph Algorithm Efficiency Test

4.3.1 Overview

This test focused on evaluating the efficiency of graph algorithms within Neo4j. Graph algorithms are a fundamental piece for any graph database, as they are responsible for extracting meaningful insights from complex network structures, thus their performance is critical in a graph database environment.

4.3.2 Methodology and Execution

For this test, the **PageRank** algorithm was selected, known for its ability to measure node influence within a network. The algorithm was executed to identify the most influential users in the Stack Overflow community, based on their interactions within the network. The Cypher query executed utilized the GDS (Graph Data Science) library to stream PageRank results, targeting the **'User'** nodes and **'ASKED'** relationships, and is as follows:

```
CALL gds.pageRank.stream({
 nodeProjection: 'User',
 relationshipProjection: 'ASKED'})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).uuid AS user, score
ORDER BY score DESC
LIMIT 10;
```

4.3.3 Results and Observations

The execution of the PageRank algorithm was monitored for its runtime efficiency and the amount of I/O it used, particularly focusing on larger subsets of the dataset. Notable increases in execution time were observed as the dataset size grew, indicating a higher computational load for larger graphs. Furthermore, the **PROFILE** output showed significant database hits and memory usage, particularly in dense network areas, which highlighted the resource-intensive nature of graph algorithms like PageRank. Finally, it should be noted the algorithm efficiently utilized the indexes where applicable, which aided in faster node retrieval times.

The Graph Algorithm Efficiency Test helped to show the the power and resource demands of graph algorithms in Neo4j. While effective in extracting valuable insights, these algorithms require considerable computational resources, especially with large and complex datasets like Stack Overflow. Understanding these trade-offs is very important for optimizing performance in graph-centric applications, as this test goes to show that while graph databases come with their benefits, they also come with their drawbacks.

4.4 Virtual Graph Creation Test

4.4.1 Overview

Virtual graph creation within Neo4j offers a dynamic approach to analyzing relationships and patterns in graph data. This test attempts to examine the performance implications of creating virtual graphs using the APOC library. The APOC library (Awesome Procedures On Cypher) is an essential toolset for working with Neo4j. It extends the capabilities of Neo4j by providing a collection of pre-built procedures and functions that cover a range of functionalities not natively available in Cypher, Neo4j's query language.

4.4.2 Methodology and Execution

The test involved using APOC procedures to create virtual graphs representing specific aspects of the Stack Overflow data. The goal was to assess the time efficiency and resource consumption of these operations. The query used for this purpose is as follows:

```
CALL apoc.nodes.group(['User', 'Question'], ['ASKED']) YIELD
    ↪ virtualGraph
RETURN virtualGraph;
```

This query creates a virtual graph that groups '**User**' and '**Question**' nodes based on the '**ASKED**' relationship, essentially condensing the complex network into a simpler, more manageable form.

4.4.3 Results and Observations

The performance of the virtual graph creation was evaluated in terms of execution time and the impact on system resources, with a focus on varying data sizes. The creation of virtual graphs exhibited a linear increase in execution time corresponding to the size of the data subset. As the complexity and volume of the data grows, this operation will undoubtedly take much longer. Additionally, the operation demonstrated moderate resource consumption, with CPU and memory usage increasing in proportion to the complexity of the data being processed.

In conclusion, the creation of virtual graphs proved to be a powerful feature for dynamic data analysis, allowing for flexible and on-the-fly reorganization of the graph structure. However, the performance cost associated with this flexibility became more apparent with larger and more complex datasets. While virtual graph creation is a great tool to use for exploring and analyzing graph data in new ways, it requires consideration of the dataset size and complexity due to its direct impact on execution time and I/O demand. In the context of large datasets like Stack Overflow, the use of virtual graphs can provide significant analytical benefits, but with an associated increase in computational demand.

4.5 Data Scalability Test

4.5.1 Overview

A crucial aspect of database performance evaluation is understanding how the system scales with increasing data volumes. This test will attempt to examine the scalability of Neo4j, particularly how data volume impacts query execution time and system resource utilization.

4.5.2 Methodology and Execution

To assess scalability, a set of queries was executed while incrementally increasing the size of the Stack Overflow dataset. The queries that were used are as follows:

High View Count Questions Query

This query counts the number of questions with more than 1000 views, testing read operations against growing data.

```
MATCH (q:Question) WHERE q.view_count > 1000 RETURN COUNT(q);
```

Popular Tags Query

This query retrieves the count of unique users who have provided high-scoring answers, testing both the read operations and relationship traversal efficiency.

```
MATCH (u:User)-[:PROVIDED]->(a:Answer) WHERE a.score > 10 RETURN
    ↪ COUNT(DISTINCT u);
```

4.5.3 Results and Observations

The performance of these queries was monitored in scenarios with varying dataset sizes, enabling an assessment of how well Neo4j manages larger volumes of data. As the dataset grew, there was a noticeable increase in the response time for both queries. The growth in execution time was greater in the query involving relationship traversal, highlighting potential scalability challenges in more complex query scenarios.

The Data Scalability Test provided valuable insights into the performance characteristics of Neo4j as data volumes increase. It highlighted the importance of considering data growth in the design and tuning of Neo4j databases. For applications like Stack Overflow, which can accumulate large volumes of data over time, understanding and planning for scalability impacts is essential to ensure sustained performance and resource efficiency.

5 IGNORE BELOW

These are different types of citations: a book citation [2], a paper citation [3], and a website citation [1]. You can also cite a website using a footnote.[1] This is how you can refer to a figure: Figure **??** shows the cover of a book about databases by Rick Hull et al. You can also refer to a section, e.g., Section **??**.

References

[1] Apache Spark. https://spark.apache.org/. Accessed: 2022-09-01.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[3] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996.

[1] https://spark.apache.org/