Neo4j and Graph Databases

Josh Redelinghuys
Department of Computer Science
University of Cape Town
South Africa
RDLJOS002@myuct.ac.za

ABSTRACT

Students and staff at UCT face difficulties when registering for degrees with complex requirements. To address this, a graph database is proposed to implement fact-checking (to ensure the correctness of the courses registered by students required for a degree) and to provide indications of how similar students faired in the past. Neo4j is a new (± 12 years old) graph database management system that has been shown to manage highly-connected data far more efficiently than relational databases. Neo4j handles specific queries on data much faster than relational databases but struggles with very large datasets due to memory consumption. Neo4j can be used effectively to perform decision making and exploit the relationships between data to make meaningful recommendations by using the similarities between data.

1 INTRODUCTION & MOTIVATION

This project aims to solve some of the problems faced by students during the registration and early stages of their Computer Science degree at UCT. To qualify for a degree in computer science, registration in several compulsory and optional courses are required in varying combinations. This creates much confusion among new students who are unfamiliar with registering for courses and electives, resulting in administrative complications for both the student and staff. To resolve this a graph database is proposed to perform fact-checking (to ensure a student has registered for the correct courses) and provide some feedback to the student regarding the performance of other students who registered for similar courses in the past. Graph database software packages like Neo4j are still new and many of its applications are still to be explored.

This review is required to serve as an investigation of more modern database technologies in the possibility of using some of the discussed techniques to address the aforementioned problem.

Relational databases have been the traditional data store for many years, but as modern data grows in volume and interconnectedness, newer, more flexible models are required [21]. To illustrate the need for NoSQL and graph databases, a brief introduction to the relational database and its limitations will be provided. Next NoSQL databases and their applications will be used to frame the motivation for graph databases. Further, the design, implementations and advantages of graph databases will be detailed to serve as a foundation for the elements and applications of Neo4j outlined.

The features and advantages of Neo4j will be contrasted with the performance of real-world applications of the software. This shall be used to understand the strengths and limitations of Neo4j discovered in applied problems. Finally, some case studies will be explored to determine the best procedures to follow when checking constraints and developing suggestions in Neo4j.

2 RELATIONAL DATABASES

2.1 Design & Applications

For many years, the standard approach to storing and accessing large volumes of data was through Relational Database Management Systems (RDBMS) [21]. This approach is widely adopted and best suited to structured, aggregated data [25]. Aggregate data (numerical and non-numerical information collected from multiple sources in their lowest level of detail) is most effectively stored in RDBMS due to the natural correlation between the structure of the database and the data itself. RDBMS maintain their data in tables consisting of rows and columns, where items in rows constitute unique data objects and the corresponding values in the columns are the attributes of that data. Data which are inherently structured are then most logically stored in organised and similarly structured tables [25].

Data stored in RDBMS are governed by a schema - a set of predefined rules which specify the structure of the tables storing the data and the relationships between them. Rows, also known as records, are uniquely identified by their primary keys which are fundamental in defining the relationships between tables. Data stored in these tables are often *normalised* to various degrees (when the database is altered to remove repetitions of data). When a relational database is queried *JOINs* are often required; this process compares columns from two separate tables and produces a new table of results filtered on a query criterion [40].

Relational databases have been the most popular data store for decades due to several benefits. As the relational model has existed for many years, it can be considered a mature system - one which is highly tested and reliable. Because of rigorous usage over the years, the stability of RDBMS has been confirmed, thereby supporting its high adoption rate [21]. Further, these systems (although easily optimised for small applications) can grow to large multi-user applications, and the rich support system which has developed over time is highly advantageous for distributed databases. Over time, many improvements have been made to the relational software packages including data integrity, security, the flexibility of deployment, and ease of programming in SQL (the Standard Query Language for RDBMS) [38].

2.2 Limitations

Recently, the growth in the amount of data produced daily and the interconnectedness thereof have revealed weaknesses in the relational model. IBM reports that 90% of all the data in the world (from sources such as imagery, transactions and social media) was generated in the last two years [1]. As a result, relational databases are required to distribute their data on multiple servers. This is not only an expensive solution but impacts both the

processing of queries (JOINs across multiple servers do no perform well) and the database schema (redesigning the schema when the database expands is not a simple task) [21].

In general, the performance of RDBMS decreases as the volume of data stored increases [26]. This degradation is explained by the nature of JOINs (the more data stored, the more comparisons required) [27]. JOINs create Cartesian products of all possible combinations of rows specified in the query and eliminate results which do not match a specified condition. In a database with one million records, a query containing five JOINs will result in billions of results to compare which is infeasible [39]. To reduce the number of JOINs required, redundancy can be introduced into the database (thereby de-normalising the data), but this complexity can quickly become unmanageable [25].

This problem becomes more serious as data becomes more interconnected [25]. The rapid increase of data volume and the relationships which exist between them is not efficiently managed by RDBMS [17]. Highly-connected data is often *semi-structured*, where not all columns of a row contain a value. To accommodate this, a relational model may be extended to incorporate new, optional attributes (in the form of additional columns); this will undoubtedly produce gaps in the table (see Figure 1). Not only does this complicate the schema required (and potential redesigns of the database), but semi-structured data, which contains many-to-many type relationships, is often cyclic and 'deeply-nested' which further complicates querying [2].

Moreover, as the types of data collected evolves, the rigidity of RDBMS simply cannot support the growing complexity of the data and evolve to adhere to the requirements of storing and querying such data [27].

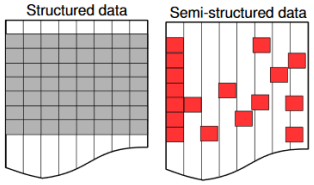

Figure 1: Structured & Semi-Structured Data [36]

3 NOSQL DATABASES

To address the limitations of traditional relational databases, the NoSQL model was developed [26]. NoSQL or 'Not Only SQL' attempts to solve the problems of exponential growth in the volume of data produced and the complexities and interconnectedness of this data. This is done by avoiding the typical tabular structure of RDBMS. Although relational databases may remain suitable for many applications, a NoSQL system would be more suited to data with the following characteristics [38]:

- Interconnected data with multiple many-to-many relationships.
- Data which are continuously evolving (and requires regular schema changes).
- Unstructured/ Semi-structured data which, when stored in an RDBMS, produces large sparse tables with many columns only used by some rows.
- Data which are stored across multiple servers and requires concurrent access [39].

NoSQL databases can be organised into 5 categories [3]:

1. Wide-Column stores which store similar data in continuous columns.
2. Document stores, a schema-less database which stores its records as semi-structured documents.
3. Key-Value stores, a schema-less mapping of key-value pairs for fast indexing.
4. Object-Oriented Databases, where data items are stored as objects (thereby encapsulating both OOP and database features) retrieved by a unique identifier.
5. Graph Databases, where data is represented as a graph consisting of nodes and edges which represent objects and the relationships between them.

These non-relational databases are highly scalable [26] and are focused on the efficient processing of queries [15]. To achieve this improvement of data processing in NoSQL databases, Brewer proposed the CAP theorem [32] as an answer to the lack of ACID characteristics in NoSQL databases [21]. The ACID properties of Atomicity (only successful transactions completed atomically will change the database), Consistency (no contradictions exist in the database), Isolation (many simultaneous operations will be executed sequentially), and Durability (no data will be lost due to database crashes) are responsible for the reliability of RDBMS [39]. The CAP theorem posits that, realistically, a distributed database system may only have two from the following properties: Consistency (C) all clients have access to the same version of the data, Availability (A) all clients have access to the data, Partition tolerance (P) where the system retains its functionality when spread across multiple servers [32].

Other advantages NoSQL has over RDBMS are the availability of multiple alternative models to best represent the data to be stored [26], as well as price advantage of open source software which supports NoSQL [15].

Contrarily, some NoSQL databases are limited by their lack of ACID support, their immaturity (lack of sufficient development and testing), and, in some cases, the lack of SQL as a standard cross-platform language [26].

4 GRAPH DATABASES

As this research project will focus specifically on Neo4j, a graph database management tool, Graph databases (GDB) shall be the main NoSQL database type discussed.

4.1 Design & Applications

Data stored in GDBs are represented as Directed Acyclic Graphs (DAG) consisting of nodes and edges between the nodes. Data typically stored as records in RDBMS are contained in nodes. The relationships between records are represented by the edges between nodes - these are directed and labelled connections between nodes which may have their own attributes. The attributes describing nodes and edges are known as their properties whereas their labels are used to classify the types of nodes and edges represented [38]. See Figure 2 of a simple example of data typically stored in a graph. GDBs are schema-less meaning the design of nodes and edges do not have to be pre-defined. Nodes and edges only contain the attributes that they possess i.e. if a record does not contain a value for some possible attribute, it will not be stored in that node [29].

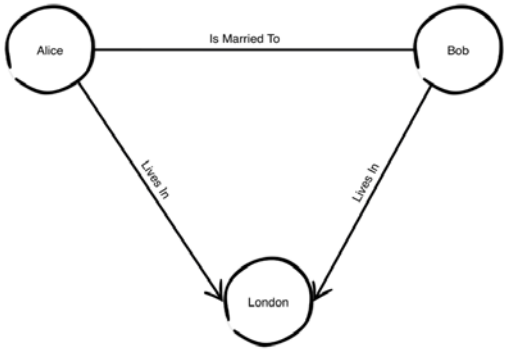

Figure 2: A graph representation of data [20]

GDBs do not store a global adjacency matrix containing all the connections between all the nodes, rather each node only tracks which other nodes it is connected to. However, a global index exists which is used to locate nodes in queries. Queries in GDBs are facilitated through *traversals* along edges between nodes. Multiple traversals of a graph database are equivalent to recursive JOINs in a relational database. Instead of comparing multiple columns in many tables in RDBMS, only the necessary edges between nodes are traversed in GDBs thereby avoiding processing unrelated data [25].

GDBs are not suitable to all types of data and should be used in specific cases to make the best use of their features. Most commonly, GDBs are used for dynamic, semi-structured, highly-connected data which is constantly evolving [29]. In GDBs the relationships between data are as important as the data itself [26] and as such, a GDB is most effective at detecting patterns, correlations and recommendations [25].

GDBs are frequently used in social networking and recommendation applications [26] as they are most useful for analysing the relationships between data. Of all NoSQL databases, GDBs are the only type which places a large focus on the relationships between data. This has intuitive benefits, as highly-connected data can easily be 'modelled' by drawing a graph - GDBs provide a direct mapping between how the data is structured and the database itself [15].

4.2 Graph Traversals

Querying data in GDBs is achieved through graph traversals, which are totally unlike SQL JOIN statements in relational databases. One motivation to use GDBs over RDBMS is the faster query speed, achieved through these traversals. Traversals only move between nodes via edges and therefore only operate on relevant data; this is unlike JOIN operations which first consider the entire data set of relational tables. Traversal queries start at an initial node (determined by the global index) and follow the necessary edges (determined by the query), visiting other nodes (and following their edges) until the constraints of the query no longer apply. Traversals will only visit those nodes which are in some way connected to the starting node. All nodes visited during the traversal are collected as results of the query [20].

T. Abedrabbo et al. [39] best illustrated how a traversal differs computationally to SQL JOINs through the following example. Two methods can be used to count the number of people within 15 feet of a person in a crowd. Firstly, the person can observe all the people within 15 feet of themselves and count the number. Secondly, the person can count everyone in the entire crowd and then subtract those who are not within 15 feet. The first method demonstrates the efficiency of a graph traversal in GDBs whereas the second method represents how, for interconnected queries, SQL JOINs are inadequate.

The nature of graph traversals ensures that the performance of graph queries remains constant and is unaffected by the size of the database. Although some queries will inevitably visit more nodes than others (and execute in more time) the performance remains independent of the total database size [29].

4.3 Theoretical Advantages of GDBs

As discussed in Section 2.3.4, graphs are not suitable for all data types and queries but do still exhibit clear advantages over relational alternatives. The following applies to GDBs and Neo4j particularly.

For highly connected data, GDBs handle their data faster. Graph databases query data through traversals which make use of powerful graph algorithms to substantially improve the performance of data retrieval when compared to relational queries [29]. Neo4j claims that a query depth of 100,000 (100, 000 relational JOINs) is easily computed in seconds, something RDBMS are incapable of [36]. Traversals are implemented using pointers instead of the lookups used in RDBMS. These pointers are regularly updated and ensure that fast performance is not affected by the size of the database. Pointers are more efficient as they provide a direct link to the data required, instead of some searching which is inevitable in indexing [27]. A global index of nodes exists in GDBs to determine the starting node of a query - this completes in O(logn) time [25].

The growth of a database is far better managed by GDBs. Not only is the time taken to execute graph queries independent of the database size [25], but the absence of a rigid pre-defined structure allows the database to evolve over time. GDBs require no schema to define and order the data stored and therefore the data can vary and restructure as necessary. New relationships can be added without the need for re-normalisation or the creation of intermediary tables as with RDBMS. This allows GDBs to scale with greater ease than relational tables, for which the performance of JOIN operations degrades with data size and the restructuring tables are not simple [27, 29].

Graphs are a logical, natural and strong representation of data which is easier to understand than a series of tables. Specifically, for many-to-many relationships, graphs are a more user-friendly approach to data storage. Without having to perform complex JOIN queries across multiple tables, retrieving data can be made much simpler to code when working with graphs [20].

4.4 Performance of Graph Databases and Relational Databases

Many experiments which compare the performance of graph databases (using Neo4j) and relational databases (commonly MySQL) have been conducted.

As expected, graph databases excel at traversal-based queries because of their design. Traversals centred around a *query depth*, the number of consecutive edges traversed, are what graph databases like Neo4j were designed to do. For example, finding all nodes related to all nodes immediately connected to an initial node constitutes a query of depth 2. Consequently, Vicknair et al. [38] concluded through their experiment with structural queries (finding all nodes without any edges and queries counting the number of nodes reachable at varying depths) that Neo4j was considerably faster than the relational implementation. Similarly,

Abedrabbo et al. [39] show that for query depths greater than 2, Neo4j significantly outperforms its SQL competitor, which at query depths of 5 could not complete execution (due to the computational expenses of multiple JOINs). For single JOINs, RDBMS and GDBs share similar performance [39]. Hölsch et al. [13] further found that for queries involving cyclic relationships (thereby involving many repeated JOINs of the entire database) and pattern-matching queries (finding all nodes with specific labels) were faster in graph databases than RDBMS.

Hölsch et al. [13] found that their analytical queries on databases (finding the number of ingoing and outgoing edges per node, all pairs of nodes with an edge between them, the shortest path between all possible other nodes, and the longest shortest path between all nodes) executed in less time in their relational model than the graph implementation. As these analytical queries accessed all nodes it was hypothesised that the faster times were a result of the advanced disk and memory management protocols implemented in relational databases (which are absent in Neo4j). Likewise, Neo4j was outperformed by RDBMS in queries which searched for numerical attributes in nodes. Neo4j indexes node attributes as strings, not integers and the overheads of converting between representations slowed the execution time [38].

5 NEO4J[1]

Many software tools such as AllegroGraph, DEX, Filament, G-Store, HyperGraphDB, InfiniteGraph, Neo4j, Sones and vertexDB implement graph databases [3]. Domingues-Sal et al. [6] found through extensive benchmark tests that DEX and Neo4j were the most efficient databases. As the market-leading software [13], Neo4j will be the main focus of this review.

5.1 Structure

Neo4j is designed for network-oriented, semi-structured data, which is most naturally represented in a graph structure and which stores many varying, optional attributes [36]. Data in Neo4j is not stored on a relational backend (i.e. Neo4j is not a layer on top of a traditional relational table), but instead, each object is stored *natively* [6]. A native graph database backend is a storage structure which is optimised for graphs [3]. For example, nodes and their associated relationships are stored close to each other on disk. Similarly, the traversal API and query language, Cypher, are stored within proximity of one another. The Neo4j native graph backend is responsible for the speed of graph queries [10].

Nodes and the edges (relationships) between them form the *node space* which represents a collection of data. Neo4j is written in Java and therefore the node space is stored in an object-oriented approach. Each node contains an ID and a list of attributes (in addition to methods which access them). Neo4j is optimised for highly-connected data hence, relationships in Neo4j are objects themselves (see Figure 3). Relationships also contain an ID, direction and attributes (and corresponding methods) [36]. The inclusion of attributes contributes to the performance of accessing the data in that node or edge [3].

[1] 'Neo' has been interpreted as an acronym for 'Network Engine for Objects' but its name originally took inspiration from the character Neo in *The Matrix* who rebelled against the matrix (i.e. a table), thus playfully nodding to the relational database model. The '4j' stands for 'for Java'.

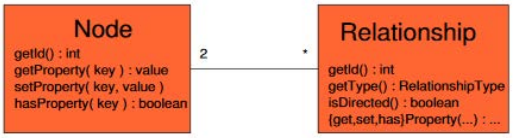

Figure 3: Node and Relationship in Neo4j [36]

Neo4j is classified as a *navigational database* where the results of queries are determined by following a path from a given start node. Traversals in Neo4j consist of the starting node, the relationship to traverse, the criteria to determine if a node is a result, and a stopping criterion. Result selection criteria are usually simple Boolean evaluations and stopping criteria stipulate a traversal to a certain depth (or a traversal of the entire network). Neo4j implements this in their traverser framework (see Figure 4) which considers a traversal as an object itself [36].

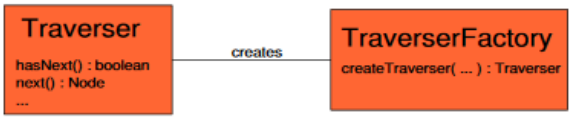

Figure 4: Traverser in Neo4j [36]

5.2 Features & Functionalities

Features that are standard with most database management tools such as a user interface, a query language and optimiser, and an Application Programming Interface (API) are all present in Neo4j [3].

Data Storage. To maintain enormous quantities of data, Neo4j supports external storage in addition to the files stored in main memory. Neo4j has an upper limit on the graph size of 10^{10} nodes. This is enough to support most graph problems [29]. Further, Neo4j supports *sharding* (dividing a database into smaller, more manageable sections) for distributed sharing of data over a network. Neo4j indexes their data for efficient data retrieval [15].

Data Scalability. When scaling (and sharding) a database, spreading highly connected nodes over multiple machines degrades performance. Neo4j implements a *master-slave* protocol whereby data stored on a master machine is automatically copied to other (slave) machines. Data is only written to the master, but data can be read from any of the slaves, allowing concurrent access of the same data across a network. Additionally, Neo4j uses two levels of caching to improve the performance of data retrieval [29].

Data Integrity. Neo4j sets itself apart from other NoSQL databases as it offers full ACID support. Many other implementations that adhere to the CAP theorem are adopted tentatively in industries where the reliability of transactions is fundamental [21]. Neo4j places a specific focus on its database management and as a result is fully ACID compliant just as relational databases are [39].

Data queries. Several methods exist to retrieve data from Neo4j: API calls, REST framework, and query languages (Cypher, SparQL, Tinkerpot, Gremlin) [11]. These can support the following query types [3]:

1. Adjacency queries, the determination of adjacent or related nodes.
2. Reachability queries, the determination of traversable paths between nodes.
3. Pattern matching queries, the location of subgraphs which match a specific pattern.

5.3 Cypher

When using the Neo4j Java API, straightforward queries can easily grow in complexity and become more challenging to code [36]. Neo4j's query language Cypher allows for more simplified statements to access data stored in Neo4j. Cypher is a pattern-matching query language i.e. Cypher statements describe what is to be retrieved from the graph [39]. Cypher is like SQL (the most popular relational query language) and uses the following basic keywords (see Table 1):

Table 1: Cypher query statements

Keyword	Description
`CREATE`	Used to declare nodes and relationships.
`START`	Used to specify the starting node of a query. This can be omitted and Neo4j will infer a start node from the MATCH clause.
`MATCH`	Used to match nodes and edges to a specified pattern, usually based on a relationship between nodes.
`WHERE`	Used to filter results.
`RETURN`	Used to specify what results should be returned from a traversal.

Cypher queries return 3 artefacts: the variable name of the value returned by the query, the values returned by the query, and a modification summary including how much data was returned, how the database was modified, and the time taken to execute the query [20].

For example, to query a network of people to find all friends of a given person, the following Cypher query can be used. This query specifies the start at node 1 (person1) and returns all nodes (person2) who are connected to person1 by the `[:IS_FRIEND_OF]` relationship.

```
START person1 = node(1)
MATCH (person1)-[:IS_FRIEND_OF]-(person2)
RETURN person2;
```

This may return the following output:

```
+---------------+
| person2       |
+---------------+
| "Bob"         |
+---------------+
1 row
15 ms
```

Holzschuher and Peinl [14] found that, in addition to Cypher performing well in most queries (when compared to other Neo4j query languages), subjectively, Cypher was the easiest language to use due to its intuitive nature and similarities to SQL.

5.4 Graph Traversal Algorithms Used

To execute its queries, Neo4j implements *Breadth-First-Search* and *Depth-First-Search* algorithms to traverse its graph data. When traversing a graph, the number of *hops*, a movement from one node to a connected node, should be minimised. A query executes faster, and less memory is used when the number of hops is minimised. During the traversal of a graph, the correct edge that will likely complete the query faster should be traversed next. Deciding which algorithm, Breadth-First-Search (BFS) or Depth-First-Search (DFS), will perform the best depends on the query [20, 39].

The DFS algorithm first visits all the *children* of a node before progressing to a node with unvisited children. Children nodes are those which descend from a node. *Siblings* are those unconnected nodes which are 'on the same level' as a node. In Figure 5, nodes 2, 3 and 4 are children of node 1. Nodes 2, 3 and 4 are siblings. In a DFS the nodes in Figure 5 are visited in the order 1 → 2 → 5 → 6 → 3 → 7 → 4. The nodes on the left side of the graph are visited far before the other nodes closest to node 1 (nodes 5, 6 and 7 are visited before node 4 which is closer to node 1) [35]. DFS is the default traversal strategy in Neo4j used in Cypher queries. When accessing Neo4j within Java using the Traversal API, the search algorithm can be changed to BFS in the query [39].

Whilst DFS moves through a graph vertically, BFS takes a horizontal approach by visiting all sibling nodes before children nodes. In Figure 5, when a BFS is used, the nodes are visited in the order 1 → 2 → 3 → 4 → 5 → 6 → 7. The nodes closest to the start node are visited first. The BFS algorithm has been widely adopted as a base for many other graph traversal algorithms, including the shortest path calculation between nodes [4].

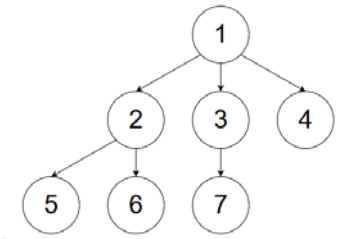

Figure 5: An acyclic graph or tree

A BFS approach is more appropriate when querying nodes closely connected to the starting node, where DFS is optimal for querying data in the left side of a graph. The choice of algorithm directly impacts the number of hops required to complete a traversal. A DFS may prove inefficient when result nodes are located close to the start node and BFS may traverse the entire graph before finding a node in close proximity to the start node [19].

The choice of the algorithm also impacts the storage required to execute a query. Traversals must record what nodes have been visited and what nodes are still to be visited. When using DFS, entire subgraphs are completely traversed and therefore can be marked as 'visited' and will not consume further memory. When using BFS, many nodes and their children are held concurrently in memory as they wait to be visited, thereby requiring more memory storage [39].

5.5 Neo4j Limitations

Java uses its *Heap Space* structure to allocate memory during run time. When new objects are created, they are stored in the Heap; Java *Garbage Collection* is used to remove objects from the Heap that are no longer needed. When Java Garbage Collection occurs all threads (used for parallel processing) are paused to safely remove objects from the Heap [5]. Because Neo4j was created using Java, the Heap structure and Garbage Collection are used when accumulating results during a traversal query. Garbage Collection can affect the performance of a query as the threads responsible for fast processing are continuously interrupted [8].

The Heap can be assigned a size (commonly between 1 and 32 GBs); Neo4j themselves state that trial and error is required to determine the optimal Heap size for a specific graph. If the Heap is too small this can increase the number of "Out of Memory" errors which can halt performance by "as much as two orders of magnitude" according to the Neo4j documentation [37]. This problem is experienced with common algorithms (such as BFS described in Section 2.4.4) applied to large datasets.

Whilst Neo4j is adamant that their database can manage enormous data sets (hundreds of thousands of relationships) with ease [36], empirical data has shown otherwise. Miler et al. [24] found that with the improved performance of their Neo4j database, it consumed between 20% and 75% more storage than the same data stored relationally. The same results are confirmed by Vicknair et al. [38] who found their Neo4j implementation to be almost twice the size of the relational implementation. Some queries on super-massive data sets (hundreds of millions of edges) could not only not complete on Neo4j, but also required the most memory (compared to other software packages). It is possible to store these large graph databases, but the memory required to execute the queries like a BFS require too much additional memory which can be infeasible in Neo4j [23].

Hoksza et al. [12] further found that queries searching for subgraphs in their cyclical graph containing 14 million nodes 38 million edges (roughly 3GB's) were infeasible. Pobiedina et al. [28] showed positive performance results with Neo4j on large acyclic graphs, but performance on massive, cyclic data sets was far worse than other databases. However, Jouili and Vansteenberghe [18] found that Neo4j produced the best results for shortest path (using BFS) calculations when compared to other GDB software.

Furthermore, Neo4j struggles with arbitrary queries such as "how many of my customers over age 25 and a last name that starts with an F have purchased items the last two months?" on data. Such queries do not rely on relationships and traversals and therefore are not optimised in Neo4j [36]. Other query types, such as those which require nodes to reference themselves (in a recursive manner) are not supported by Neo4j as such relationships cannot be defined in Neo4j [29].

5.6 Case Studies & Applications

As the two main facets of this project will involve constraint-checking and recommendations based on historical data, the following case studies (using Neo4j) will be discussed.

A. Constraint Checking

To check that a set of input data meet a set of predetermined criteria, a *decision tree* is required. Although other, more complex structures are available, such as the newly developed *decision streams*, these are more suited to decisions with multiple answers (not just Yes/ No decisions) [16]. Graphs without cycles are considered trees (as seen in Figure 5) [39]. The term 'tree' is more common in decision modelling and will be used in place of 'graph'.

Decision trees are structures used to organise data into categories (such as "Meets the requirements of degree" and "Does not meet the requirements of degree" categories). A decision tree consists of three types of nodes: the root (where the decision process begins), rule or test nodes (which apply some test to determine which category the data falls into) and a leaf or category nodes (which represent a category that the data may be classified into). The root node is usually also a test node. Data are tested in rule nodes, and the outcomes of these test determine which edge to follow next. The data are continually tested, moving from one rule node to the next until it reaches an answer node [30].

Neo4j describes the procedure required to implement a decision tree on its platform as follows [31]. As decision trees are more complex, Java methods which make use of the Traversal API are required. The main Java method which executes the decision process accepts a Neo4j graph (the tree) and the data to be categorised. Beyond that, the following components are required [7]:

- The tree itself. This is created using Cypher's `CREATE` command. Rule nodes will have the attributes: expression (the condition applied in that rule e.g. age >= 21), parameter name (the name of the parameter supplied to the expression e.g. 'age') and the parameter type e.g. 'int'. Category nodes will have an ID and description. Rule nodes will be connected to other nodes by "Is True" and "Is False" relationships.
- The Evaluator. This method decides to stop traversing if an answer node has been reached or to continue down an edge determined by the expander.
- The expander. This determines where to traverse by using the expression evaluator.
- The expression evaluator. This evaluates a rule based on the expression attribute stored in rule nodes, using the data input. The result of the evaluation determines the next edge to follow.

The graph is continually evaluated using a DFS until the data is classified in an answer node.

B. Historical Results & Recommendations

The relationships between data can be exploited to make recommendations. This widely used in online-shopping and social networks where "Customers who viewed this item also viewed..." and "People you may know..." are becoming commonplace in each field respectively [20]. Similarly, historical data can be used to indicate past trends which also influence recommendations. In the context of this project, providing an "indication of how students in similar situations fared in the past" is comparable to extracting information in the same way a recommendation engine produces suggestions.

Gorakala [9] outlines the process of building a recommendation engine in Neo4j. First, find related nodes which are either mutually connected to a node or have similar outgoing/ ingoing edges. Then, by using a count of the number of similarities between the nodes as a metric of their relatedness, extract data to recommend from the most related nodes. In the case of movie recommendations: (1) count the number of high-rated movies in common between a user and all other nodes as an indication of their similarity (2) for the nodes considered similar (with a count greater than n) retrieve all other high-rated movies not seen by the user (3) recommend those movies.

Further examples detail other methods available to extract suggestable data. Vukotic et al. [39] demonstrate that finding all of the movies watched by all of the friends of a person, a rudimentary recommendation can be made. Their query makes recommendations by first identifying all the friends of a person, and then all the movies watched by those friends which the person has not yet seen. In a similar example, Lu et al. [22] specified the `MATCH` queries used to access movies that were linked through a 'directed by' relationship to recommend to a user.

Stark et al. [34] outline their process of drug recommendation using a graph of patient data. Extracting the data to recommend requires a Cypher `MATCH` statement on a specific relationship. Their algorithm accepts parameters relating to a patient's current health and makes a recommendation based on other patients with similar parameters. The search space for possible drugs to recommend was first reduced through a decision-tree style filtration where only patients with the same gender and background were considered. Singh and Kaur [33] describe how such medical data could be modelled in Neo4j to support such queries. Patients, diseases, treatments and doctors are represented by nodes and the relationships between them consist of: 'was diagnosed by', 'is affected by' and 'is treated with.'

6 DISCUSSION

Since data is currently produced at much higher volumes than 50 years ago, much investigation has been conducted into the field of NoSQL databases. As the limitations of relational databases became unavoidable, NoSQL databases grew in popularity. As a result, the performance of these various NoSQL models was of great interest to database engineers. Many surveys and evaluations have compared NoSQL databases with one another and with traditional relational implementations [11, 18, 23, 26]. The various types of NoSQL databases are all conceptually different to the rows and tables found in RDBMS and all have their own feature set that makes them appropriate for different situations. The different types of data and situations best used for the multiple NoSQL models have been discussed.

Graph databases are a popular NoSQL model that specialise in representing highly-connected data. The model of using nodes and edges results in efficient queries using graph traversals. The importance of the type of traversal selected has been discussed in addition to the theoretical advantages of speedup and scalability in GDBs [25, 39].

Many papers investigate the restrictions placed on Neo4j by Java's memory allocation; in many cases this contradicted the advertised speed-up of Neo4j. Whilst, in many comparisons (between GDB software and RDBMS software), Neo4j produced the best overall results, it was still not consistently the top performer. It is still, however, the best software option to address the research question [18].

Recommendation engines and decision trees can be implemented in Neo4j in a straightforward manner. The evaluations of Neo4j conducted on enormous sets of data indicate that queries involving the entire graph will be potentially infeasible in situations where the number of nodes in the graph is in the millions. The traversal query algorithm chosen is significant and in most cases is a DFS. As described by other implementations, to construct the code required for the constraint-checking and recommendations, the Neo4j API will likely be required in addition to Cypher [9, 20, 39].

Very little research on the viability of GDBs to implement decision trees is available. It is clear that a decision tree is not a common use case for Neo4j and although it is possible, it has not been investigated thoroughly. In addition, many papers discuss their methods of extracting meaningful data from the relationships occurring between data. These papers do not discuss how best to classify students as 'similar' which is required for this project's recommendation engine.

This research project will address the gap in the literature by evaluating the complexities of fact-checking a complex requirement list using a GDB and contributing to the discussion of using Neo4j for decision making. Also, experimentation with the relationships existing between students will determine a metric best used to assess the similarities of two students. This will in turn assist with providing feedback based on historical data, thereby determining a methodology for obtaining information about past students most relevant for current students.

7 CONCLUSIONS

7.1 The performance of relational databases is limited with interconnected data.

Relational databases are strictly confined to their schema of rows and columns. To access meaningful information about the data stored in RDBMS, queries involving the JOINs of tables are required. JOINs are computationally expensive when combining millions of rows from multiple tables, which is a frequent practice in modern, interconnected data.

For a relational database to evolve with the data it stores, the data are divided among multiple servers, de-normalised, and only partially filled (semi-structured data does not contain values for every column in a table). This further impacts the performance of the database as JOINs become increasingly inefficient with redundant data spread over servers.

7.2 NoSQL addresses the limitations of RDBMS.

To improve the performance of relational databases, the NoSQL model introduces some compromises. The CAP theorem allows for the restrictive ACID rules of RDBMS to partially disregarded. Each NoSQL database has its own use, and their high scalability and ability to evolve with the data they store has resulted in high rates of adoption.

7.3 Graph databases are designed to have excellent performance.

Graph databases organise their data in graphs of nodes (comparable to a row of a table) and edges connecting these nodes which represent relationships between them. It is logical and simple to represent many real-world data-sets in a graph. Because GDBs treat the relationships between data with the same importance as the data itself, queries on highly-connected data are better managed. Theoretically, the use of graph traversals instead of JOIN statements achieves dramatic query speedup. Traversals move along the edges of connected nodes only and therefore do not consider unrelated data as potential solutions as a JOIN statement would. GDBs are optimised to scale easily and evolve with the data it represents.

7.4 Neo4j offers many features to address the requirements of highly-connected data.

Neo4j is the most popular software package that implements GDBs. It offers full ACID support, is highly scalable, performs well for most data sets, and is easily integrable. Many languages are available to query data in Neo4j, but Cypher, being like SQL is most popular.

7.5 Graph traversal algorithms affect query performance.

The Breadth-First-Search Algorithm is used as the basis of many other graph traversal algorithms (such as the shortest path) but is not the default algorithm used by Neo4j due to its large memory footprint. The Depth-First-Search algorithm is most commonly

used in Neo4j and is optimal for querying data in the left side of a graph.

7.6 Neo4j struggles with massive data sets.

The Java Heap and Garbage Collector are part of Neo4j's backend but unfortunately can have large negative impacts on query performance. Some traversal algorithms require a lot of memory and when the Heap is not appropriately calibrated to accommodate this, Garbage Collection interrupts can slow processing. Queries on graphs that consider millions and billions of nodes and edges are unsuccessful in Neo4j.

7.7 Neo4j can implement decision trees and recommendation engines.

The 'constraint checking' component of the project can be implemented in Neo4j using a decision tree – this will allow input data to be sorted into valid and invalid categories. Likewise, to provide students with an indication of how other students faired, a recommendation engine can be built in Neo4j to access meaningful data from similar students.

ACKNOWLEDGEMENTS

This report was made possible with thanks to the teaching and support of our supervisor Sonia Berman.

REFERENCES

[1] 2.5 quintillion bytes of data created every day. How does CPG & Retail manage it? - IBM Consumer Products Industry Blog: 2013. *https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/*. Accessed: 2019-05-01.

[2] Abiteboul, S. 1997. Querying Semi-Structured Data. *International Conference on Database Theory* (1997), 1–18.

[3] Angles, R. 2012. A Comparison of Current Graph Database Models. *2012 IEEE 28th International Conference on Data Engineering Workshops* (Apr. 2012), 171–177.

[4] Beamer, S. et al. 2013. Direction-optimizing breadth-first search. *Scientific Programming*. 21, 3–4 (2013), 137–148. DOI:https://doi.org/10.3233/SPR-130370.

[5] Bruno, R. and Ferreira, P. 2018. A Study on Garbage Collection Algorithms for Big Data Environments. *ACM Computing Surveys*. 51, 1 (2018), 1–35. DOI:https://doi.org/10.1145/3156818.

[6] Dominguez-Sal, D. et al. 2010. Survey of graph database performance on the HPC scalable graph analysis benchmark. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2010), 37–48.

[7] Dynamic Rule Based Decision Trees in Neo4j | Max De Marzi: 2018. *https://maxdemarzi.com/2018/01/14/dynamic-rule-based-decision-trees-in-neo4j/*. Accessed: 2019-05-01.

[8] Gidra, L. et al. 2015. NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines. *ACM SIGARCH Computer Architecture News*. 43, 1 (2015), 661–673. DOI:https://doi.org/10.1145/2786763.2694361.

[9] Gorakala, S.K. 2016. *Building Recommendation Engines*. Packt Publishing Ltd.

[10] Graph Databases for Beginners: Native vs. Non-Native Graph Technology: 2018. *https://neo4j.com/blog/native-vs-non-native-graph-technology/*. Accessed: 2019-05-01.

[11] Guo, Y. et al. 2014. How well do graph-processing platforms perform? An empirical performance evaluation and analysis. *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*. 1, (2014), 395–404. DOI:https://doi.org/10.1109/IPDPS.2014.49.

[12] Hoksza, D. and Jelinek, J. 2016. Using Neo4j for Mining Protein Graphs: A Case Study. *Proceedings - International Workshop on Database and Expert Systems Applications, DEXA* (Sep. 2016), 230–234.

[13] Hölsch, J. et al. 2017. On the performance of analytical and pattern matching graph queries in Neo4j and a relational database. *CEUR Workshop Proceedings* (2017), 1–8.

[14] Holzschuher, F. and Peinl, R. 2013. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J. *Proceedings of the Joint EDBT/ICDT 2013 Workshops* (2013), 195–204.

[15] Hossain, S.A. 2013. NoSQL Database : New Era of Databases for Big data Analytics - Classification , Characteristics and Comparison. *International Journal of Database Theory and Application*. 6, 4 (Jun. 2013), 1–14.

[16] Ignatov, D. and Ignatov, A. 2018. Decision stream: Cultivating deep decision trees. *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI* (Nov. 2018), 905–912.

[17] J M Tauro, C. et al. 2012. Comparative Study of the New Generation, Agile, Scalable, High Performance NOSQL Databases. *International Journal of Computer Applications*. 48, 20 (2012), 1–4. DOI:https://doi.org/10.5120/7461-0336.

[18] Jouili, S. and Vansteenberghe, V. 2013. An Empirical Comparison of Graph Databases. *2013 International Conference on Social Computing* (Sep. 2013), 708–715.

[19] Korf, R.E. 2003. Depth-first iterative-deepening. *Artificial Intelligence*. 27, 1 (Sep. 2003), 97–109. DOI:https://doi.org/10.1016/0004-3702(85)90084-0.

[20] Lal, M. 2015. *Neo4j Graph Data Modeling*. Packt.

[21] Leavitt, N. 2010. Will NoSQL Databases Live Up to Their Promise? *Computer*. 43, 2 (Feb. 2010), 12–14. DOI:https://doi.org/10.1109/mc.2010.58.

[22] Lu, H. et al. 2017. Analysis of film data based on Neo4j. *Proceedings - 16th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2017*. 1, (2017), 675–677. DOI:https://doi.org/10.1109/ICIS.2017.7960078.

[23] McColl, R.C. et al. 2014. A performance evaluation of open source graph databases. *Proceedings of the first*

workshop on Parallel programming for analytics applications - PPAA '14 (New York, New York, USA, 2014), 11–18.

[24] Miler, M. et al. 2014. The shortest path algorithm performance comparison in graph and relational database on a transportation network. *PROMET - Traffic&Transportation*. 26, 1 (2014), 75–82. DOI:https://doi.org/10.7307/ptt.v26i1.1268.

[25] Miller, J. 2013. Graph Database Applications and Concepts with Neo4. *Proceedings of the 2013 Southern Association for …*. 2324, S 36 (2013), 141–147.

[26] Nayak, A. et al. 2013. Type of NOSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems*. 5, 4 (2013), 16–19.

[27] Perçuku, A. et al. 2017. Modeling and Processing Big Data of Power Transmission Grid Substation Using Neo4j. *Procedia Computer Science* (Jan. 2017), 9–16.

[28] Pobiedina, N. et al. 2014. Benchmarking Database Systems for Graph Pattern Matching. *Database and Expert Systems Applications* (2014), 226–241.

[29] Pokorný, J. 2015. Graph databases: Their power and limitations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 9339, (2015), 58–69. DOI:https://doi.org/10.1007/978-3-319-24369-6_5.

[30] Quinlan, J.R. 1987. Generating production rules from decision trees. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. 30107, (1987), 304–307.

[31] Running Decision Trees in Neo4j @ Neo4j GraphConnect 2018: 2018. *https://neo4j.com/graphconnect-2018/session/decision-trees-in-neo4j*. Accessed: 2019-05-01.

[32] Simon, S. 2012. *Brewer's CAP Theorem*.

[33] Singh, M. and Kaur, K. 2015. SQL2Neo: Moving health-care data from relational to graph databases. *Souvenir of the 2015 IEEE International Advance Computing Conference, IACC 2015*. 0, (2015), 721–725. DOI:https://doi.org/10.1109/IADCC.2015.7154801.

[34] Stark, B. et al. 2017. BetterChoice: A migraine drug recommendation system based on Neo4J. *2017 2nd IEEE International Conference on Computational Intelligence and Applications, ICCIA 2017* (Sep. 2017), 382–386.

[35] Tarjan, R. 2005. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*. 1, 2 (Jun. 2005), 146–160. DOI:https://doi.org/10.1137/0201010.

[36] The Neo Database - A Technology Introduction: 2006. *http://dist.neo4j.org/neo-technology-introduction.pdf*. Accessed: 2019-04-30.

[37] Tuning of the garbage collector: *https://neo4j.com/docs/operations-manual/current/performance/gc-tuning/*. Accessed: 2019-05-01.

[38] Vicknair, C. et al. 2010. A comparison of a graph database and a relational database. *Proceedings of the 48th Annual Southeast Regional Conference on - ACM SE '10* (New York, New York, USA, 2010), 1.

[39] Vukotic, A. et al. 2015. *Neo4j in Action*. Shelter Island: Manning.

[40] Watt, A. and Eng, N. 2012. *Database Design*. BCcampus.