UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2019

Title: A Comparison of Neo4j and MySQL for Constraint and Similarity Checking at Registration

Author: Josh Redelinghuys

Project Abbreviation: NEO4j4US

Supervisor(s): Sonia Berman

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	10
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	10
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusion	*10*	*20*	10
Aim Formulation and Background Work	*10*	*15*	10
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	0
Total marks		**80**	**80**

A Comparison of Neo4j and MySQL for Constraint and Similarity Checking at Registration

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

ABSTRACT

Students at UCT face difficulties when registering for degrees with complex requirements. To address this, a system was proposed to check the correctness of the courses chosen, and to provide indications of how students with similar grades and curricula fared in the past.

The system was built using Neo4j, a graph database management system that has been shown to efficiently manage highly-connected data, and also MySQL, a relational database. Degree violations were determined using a constraint-checking graph in Neo4j and if-statements in MySQL, while predicted performance was calculated using the grades of similar students found through graph similarity algorithms and SQL queries in each database respectively.

It was found that for small amounts of student data, graph representation was not required as the MySQL implementation performed better. However, experimentation on larger datasets showed that when implementing tree-structures, traversing a graph database was far more efficient than JOIN statements in relational tables.

1 INTRODUCTION

Student attrition in Computer Science is a concern, seeing that the IT sector in South Africa is experiencing a shortage of skilled graduates [3]. To increase the likelihood of academic success, student morale can be improved by ensuring that they enrol in the courses most suited to them. If a student is advised of errors in their planned curriculum or that that they may not cope, they may plan to change their field of study. This results in reduced student drop-out and course repeats.

To resolve the confusion and administrative complications experienced during registration, curriculum-checking was implemented in both a Neo4j constraint-checking graph and using SQL queries. Next, to provide feedback regarding the performance of other students who registered for similar courses and achieved similar grades, Neo4j similarity algorithms were used alongside more manual SQL queries.

The features and proposed advantages of Neo4j were compared with traditional SQL methods on two different databases, one containing a small set of real student data and another containing a large set of simulated data. These database comparisons were used to better understand the strengths and limitations of Neo4j and MySQL.

1.1 Research Aims

This research aimed to explore if a graph database (Neo4j) is an appropriate means of storing student data, checking the constraints of a structured degree and further, identifying similar students for grade prediction. The efficacy of graph databases in this context was determined through performance comparisons with the relational database implementation. This project served as a proof of concept, and thus the scope covers requirements of the BSc degree only and excludes timetabling constraints and graphical user interface design.

2 BACKGROUND

2.1 Graph Database Design & Applications

Data stored in Graph Databases (GDBs) are represented as Directed Graphs consisting of nodes connected by edges. Data which are typically stored as records (rows) in Relational Databases (RDBs) are contained in the nodes. The relationships between records are represented by the edges between nodes - these directed and labelled connections between nodes may too have their own attributes [23]. See Figure S1, a simple example of data as 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 are all optional i.e. if there is no value for some possible attribute, it will not be stored in that node or edge [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. Queries in GDBs are facilitated through *traversals* along edges between nodes (located using a global node index). Multiple traversals of a graph database are equivalent to recursive JOINs in a relational database. Instead of comparing multiple columns in many tables in RDBs, only the necessary edges between nodes are traversed in GDBs thereby avoiding processing unrelated data [13].

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 [20]. In GDBs the relationships between data are as important as the data itself [14] and as such, a GDB is most effective at detecting patterns, correlations and generating recommendations [13].

GDBs are frequently used in social networking and recommendation applications [14] as they are most useful for analysing the relationships between data. Placing such importance on relationships has intuitive benefits, as highly-connected data can easily be modelled in a graph - GDBs provide a direct mapping between how the data is structured and the database itself [7].

2.2 Graph Traversals

One motivation to use GDBs over RDBs is the faster query speed, achieved through traversals instead of JOINs. Traversals only move between nodes via edges and therefore only operate on relevant data; this is unlike JOIN operations which consider entire 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 collect nodes which are directly connected to the starting node as results of the query [9].

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 [20].

2.3 Theoretical Advantages of GDBs

Query traversals make use of powerful graph algorithms to substantially improve the performance of data retrieval when compared to relational queries [20]. Neo4j claims that a query depth of 100,000 (equivalent to 100,000 relational JOINs) is easily computed in seconds, something RDBs are incapable of [22]. Traversals are implemented using pointers instead of the lookups used in RDBs. Pointers are more efficient than lookups as they provide a direct link to the data required, instead searching which is inevitable in indexing [19].

GDBs are designed to effectively manage the growth of data; unlike RDBs, the size of a graph database does not affect the performance of queries [13]. As RDBs increase, the performance of large JOIN operations decreases as more data is to be considered. Further, as GDBs are not dependent on schemas, graphs can evolve and scale with the data – no redesigns are required like in RDBs because new relationships and attributes can simply be added. This is not the case in RDBs where the addition of new attributes can often include complex schema redesigns [19, 20].

Graphs are a logical, natural and strong representation of data which is easier to understand than a series of tables. Particularly 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 [9].

2.4 GDB Performance

Traversals centred around a *query depth*, the number of consecutive edges traversed, are what graph databases like Neo4j are optimised for. For example, finding all nodes related to all nodes immediately connected to an initial node constitutes a query of depth of 2. Vicknair et al. [23] concluded through their experimentation with queries which located all nodes without any edges, and queries which counted the number of nodes reachable at varying depths, that Neo4j was considerably faster than their relational implementation. Similarly, Abedrabbo et al. [24] 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). Hölsch et al. [6] further found that queries involving cyclic relationships (thereby involving many repeated JOINs) and pattern-matching queries (finding all nodes with specific labels) were faster in graph databases than RDBs.

2.5 Neo4j Limitations

For single JOINs, RDBs and GDBs share similar performance [24]. Hölsch et al. [6] found that their analytical queries on databases (finding the number of ingoing and outgoing edges per node), 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 RDBs 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 [23].

Java *Garbage Collection* can affect the performance of a query as the threads responsible for fast processing can be continuously interrupted [4]. Garbage Collection pauses all threads are to safely remove objects from the Heap which are no longer needed [2]. The Heap can be assigned a size (commonly between 1 and 32 GBs); Neo4j documentation [17, 18] states 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 [18]. This problem is experienced with common algorithms, such as breadth-first search (the basis of the shortest path algorithm), as many nodes are stored in the Heap during the search.

Whilst Neo4j is adamant that their database can manage enormous data sets (hundreds of thousands of relationships) with ease [22], empirical data has shown otherwise. Miler et al. [12] found that their Neo4j database consumed between 20% and 75% more storage than the same data stored relationally. The same results are confirmed by Vicknair et al. [23] 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 complete in Neo4j and required more memory than other software packages. Large data set storage is possible in Neo4j, but queries like a breadth-first search require too much additional memory which can be infeasible [11]. Hoksza et al. [5] further found that queries searching for subgraphs in their cyclical graph containing 14 million nodes 38 million edges (roughly 3GBs of data) were infeasible.

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?" Such queries do not rely on relationships and traversals and therefore are not optimised in Neo4j [22].

2.6 Relational Databases

RDBs maintain their data in tables consisting of rows and columns, where items in rows constitute unique data objects and column values are the attributes of that data. Data which are inherently structured are most logically stored in organised and structured tables [13].

Data stored in RDBs 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 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 [25].

As the relational model has existed for many years, it can be considered a mature system - which is highly optimised and reliable. Because of rigorous usage over the years, the stability of RDBs has been confirmed, thereby supporting its high adoption rate [10] and over time, many improvements have been made to the query optimization and memory management of relational software [23].

2.7 Relational Database Limitations

In general, the performance of RDBs decreases as the volume of data stored increases [14]. This degradation is explained by the nature of JOINs as more data are stored, more comparisons are required [19]. In a database with one million records, a query

containing five JOINs will result in billions of results to compare which is infeasible [24]. Denormalization can be introduced to reduce the number of JOINs and indexes required but this complexity can quickly become unmanageable [13].

Large volumes of interconnected data are not efficiently managed by RDBs [8, 25] as the rigidity of tables cannot support the growing complexity of storing and querying such data [19]. Highly-connected data is often *semi-structured*, where not all columns of a row contain a value. This produces gaps in the table which complicates the schema, queries and potential redesigns of the database [1].

3 UCT BSC REQUIREMENTS

A BSc major consists of **compulsory** courses which must be taken by every student and **alternate equivalent** courses which do not have to be taken by every student. Alternate equivalent courses allow a student to choose a course or course pair which may be better suited to them. For example, a student may enrol in either MAM1000W or an alternate equivalent of MAM1004F **and** MAM1008S.

To confirm the validity of a curriculum, the following aspects must be checked [26]:

1. All compulsory courses of a specific major must be taken in the curriculum.
2. Alternate equivalent course pairs must be listed together (e.g. MAM1004F & MAM1008S).
3. The counts of specific types of courses, as defined by rules FB 3, 7.1 and 7.2 in the UCT Science Handbook [26], must not violate their corresponding limits (e.g. a curriculum must contain the equivalent of at least 6 full science courses).
4. The total NQF credits of specific types of courses, as defined by rules FB 7.7 (a) (b) (c) in the UCT Science Handbook [26], must not fall below their lower limits (e.g. a degree must contain at least 420 NQF credits).

See S2.1 for the complete list of rules applied to a Science degree. Table S2.2 summarises the counts and totals to be retrieved in order to check requirements 1 to 4 listed above.

4 SYSTEM DESIGN

The Neo4j and MySQL implementations use the same procedures (represented by the flowchart in Figure S3.1) to check a student's planned curriculum and to report the performance of similar students.

The system requires the following user input:

1. the year for which the student is registering (first years would input 1, those entering second year would input 2 etc.).
2. the student's most recently completed grades (first-years enter their NBT marks and senior students enter their GPA attained in the previous year).
3. the student's **entire** curriculum (courses enrolled in from the first year to final year).
4. the student's major/s ("CSC" represents a Computer Science Major.

4.1 Database Contents

Details stored about courses were the course name, Faculty Department, Semester, Year, NQF Credit total and the associated Major[1] which requires the course. Courses taken in year 2 and 3 are known as *Senior* courses and those students not in first year are known as Senior students. The three National Benchmark Tests (NBT) and high school subjects are all considered *Subjects* and not courses.

The only student data stored was the mark they achieved in a certain course or subject. The National Benchmark Test (NBT) marks of 528 students, the average High School English, Math and Science marks of 726 students, and the UCT course marks (across all faculties, but mainly Computer Science) of 2775 students, along with the year in which the mark was achieved were inserted into the smaller Neo4j and MySQL databases.

4.2 Checking Curriculum Constraints

As soon as any violation is discovered, it is returned to the user and the system is terminated – if any one constraint is violated, the degree plan is immediately invalid and there is no point checking any further constraints.

Initially, all courses associated with the student's chosen major are retrieved from the selected database. If any compulsory courses are not listed in the student's plan, then those missing courses are output to the student. If any alternate equivalent course/ course pair is not listed, the missing course is returned to the user.

Next, each course in the planned curriculum is queried to generate a list of count and total values outlined in Table S2.2. These course counts and NQF totals are checked against their limits and any values which are too large or too small are reported.

In the case that a curriculum meets all the requirements of the degree the system produces the output in S3.2.

4.3 Identifying Similar Students

Firstly, all students who are enrolled in similar courses as those listed in the student's planned curriculum are found. Within that group of students, those who achieved marks similar to the marks input by the user are found. The average GPA (and corresponding standard deviation) achieved by those similar students in the following year are reported to the user as an indication of how similar students fared with their degree plan. So, for a student entering second-year, the average second-year GPA achieved by similar students would be returned as a grade prediction.

The predicted grade output for a first-year with NBT marks 65, 75, and 71 majoring in Computer Science and Statistics can be seen in S3.2.

One student is "similar" to another if:

1. they are enrolled in at least 50% of the same courses. If student 1 enrolled in [CSC1015F, CSC1016S, MAM1000W] then student 2, enrolled in [CSC1015F, CSC1016S, BIO1004F], is considered similar because the number of common courses between them is 2 (which is greater than 50% of the total number of courses student 1 enrolled in).

[1] Lists of majors and their associated required courses are not available digitally, therefore the information required was collected from the Science Faculty handbook [26]. Only the requirements of a Computer Science major were stored.

2. they achieved similar marks in the previous year. In the case of first-year students, the NBT Academic Literacy, Math and Quantitative Literacy marks are compared and in the case of senior students, the GPA attained in the previous year are compared. Any mark, m_1, is considered "similar" to another mark, m_2, if m_2 lies in the range $[m_1 - 5, m_1 + 5]$. So, 62% and 67% are similar, but 62% and 56% are dissimilar.

Due to the small amount of student mark data available, a course intersection of 50% was chosen to allow for acceptable cluster sizes. Through inspection, it was found that course intersections greater than 50% can lead to student clusters of size 0 which cannot be used in further prediction. Similarly, decreasing the total 10% similar mark range can result in student clusters of size 0 which also cannot yield any predictions.

4.4 Neo4j Similarity Algorithms

Two types of similarity algorithms are required in Neo4j, one to compare lists of courses and another to compare marks. The Neo4j Jaccard and Overlap similarity algorithms calculate similarity by comparing sets and the Cosine, Pearson and Euclidean distance algorithms calculate similarity by comparing specific data points [15].

4.4.1 The Jaccard & Overlap Algorithms

The Jaccard and Overlap similarity (equation (1) and (2) respectively) of two identical sets A and B is 1, and is 0 for totally dissimilar sets. Consider set A of courses [CSC1015F, CSC1016S] and set B of courses [CSC1015F, CSC1016S, MAM1000W]. J(A,B) = 0.66 and O(A,B) = 1.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \quad (1)$$

$$O(A,B) = \frac{|A \cap B|}{\min(|A|,|B|)} \quad (2)$$

Because both algorithms find the intersection between two sets, only exact matches contribute to determining similarity, which is best suited to determining the similarity of course lists. If either algorithm were used to compare marks, the two sets [68, 78, 88] [69, 79, 89] would have a similarity of 0, despite being almost identical.

Furthermore, the Jaccard algorithm focuses on the ratio between the intersection and union of two sets, therefore generating 'stricter' similarity results. The Overlap algorithm is more lenient in its definition of similarity by dividing only by the size of the smaller set. If |A| is 10, |B| is 1 and A ∩ B is 1, then min(|A|, |B|) is 1 and |A ∪ B| is 10. The Overlap algorithm would consider sets A and B identical (similarity = 1), but the Jaccard similarity would be 0.1, a more accurate representation of course list similarity. Therefore, the Jaccard algorithm was selected to compare courses.

4.4.2 The Euclidean Distance, Cosine and Pearson Algorithms

The Euclidean distance (E), Cosine (C) and Pearson (P) algorithms are defined by equations (3), (4) and (5) respectively. Each algorithm accepts two sets of non-null values, A and B, of equal size (n). A Euclidean distance of 0 between A and B indicates that the sets are identical - the greater distance the less similar. For both Cosine and Pearson algorithms, values range between -1 and 1, where -1 is perfectly dissimilar and 1 is perfectly similar. Each algorithm contains operations on pairs of numbers, A_i and B_i, from each set. This is, therefore, unsuitable for comparing the similarity of courses.

$$E(A,B) = \sqrt{(A_1 - B_1)^2 + \cdots + (A_n - B_n)^2} \quad (3)$$

$$C(A,B) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \quad (4)$$

$$P(A,B) = \frac{\sum_{i=1}^{n}(A_i - \bar{A})(B_i - \bar{B})}{\sqrt{\sum_{i=1}^{n}(A_i - \bar{A})^2 (B_i - \bar{B})^2}} \quad (5)$$

For Science degrees, much variation exists between the courses each student has enrolled in. As a result, finding multiple students who achieved similar marks in an identical set of courses is too rare. Therefore, GPA is used as an average indicator of a student's performance – GPA averages the marks of all courses taken in a year thereby accounting for the variation in courses taken.

The issue of varying course enrolments between students is further compounded by the fact that all algorithms (E, C, P) compare two sets of equal size. To compare two sets of marks P and Q of lengths 6 and 2 respectively, set Q must be extended by 4. This is managed by the Neo4j `coalesce` function which extends the set by the required number by repeatedly inserting the first non-null value in the list. If Q contains the marks [54, 81], it would be extended to contain to [54, 81, 54, 54, 54, 54] before being compared to set P. This is a less accurate representation of the performance of student Q, whose GPA before being extended was 67.5% but is 58.5% after extending. To avoid this, the GPA of senior students is used as the single value to determine similarity of senior student marks.

This is not an issue when comparing NBT marks of students entering first-year as the size of a set of NBT marks is exactly 3. In the rare case that a student does not have all 3 marks, using `coalesce` to extend the set has less impact as a set would be extended by 1 in the worst case. If student R had mark N only, the set would be expended to [N, N, N] and if student S had marks [N,M] the set would become [N, M, N].

When comparing two GPAs, 65 and 69, E(65, 69) = 4, C(65, 69) = 1 and P(65, 69) = 0. The Cosine similarity algorithm is inapplicable to this system as when |A| and |B| are both 1, C(A, B) will always be 1 (perfectly similar) which is inaccurate. The same can be said for the Pearson algorithm where if the sizes of A and B are 1, then the numerator of the equation will always be 0. Therefore, the Euclidean distance was chosen as the best algorithm to compare GPAs of senior students.

To consistently make use of the same algorithm applied to all students, the Cosine and Pearson algorithms were not used on first-year students.

4.4.3 Euclidean Similarity Cut-off values

For first year and senior students, two separate maximum Euclidean distances where selected to reflect a total 10% range in difference between marks. Table 1 compares the Euclidean of sets of marks A and B, where $|\overline{A} - \overline{B}|$ is the absolute difference between the averages of the two sets.

When comparing single GPA values as in the first row, the average difference and Euclidean distance are equal. Thus, for senior students who are compared on GPA alone, a Euclidean distance cut-off of 5 was used.

Table 1: Euclidean Distance & Average Difference of sets A, B

A	$\overline{A}$	B	$\overline{B}$	E(A, B)	$\lvert\overline{A} - \overline{B}\rvert$
54	54	81	81	27	27
[75, 66, 71]	70.7	[82, 75, 85]	80.7	18.06	10
[80, 59, 85]	74.7	[63, 94, 91]	82.7	39.37	8

For students entering first-year who are compared on their NBT marks, a different similarity cut-off was selected. Row 3 of Table 1 shows how sets of marks which are vastly different can have an average difference less than 10%. The Euclidean distance of 39.37, however, accurately categorises the sets as dissimilar. From manual inspection of all students compared to one another, it was found that the smallest Euclidean distance between sets which have an average difference of 10% is 18.06. Using 18 as Euclidean distance cut-off, results are limited to pairs of first-years which are likely to have an average NBT mark difference in a range of 10% or less.

5 NEO4J V1 IMPLEMENTATION

This Neo4j implementation used the small, real student dataset described in Section 4.1.

5.1 Student Data Graph Design

Figure S4.1 represents the design of the Student Data graph which stores all data concerning majors, courses (taken at UCT), subjects (taken before UCT), students and marks. Student nodes are connected to course and subject nodes through edges of the type `ENROLLED_IN`. These edges contain the mark attained in a specific year in the connected course as attributes. Course nodes contain the attributes: 'Course Name', number of NQF 'Credits', the 'Faculty' that the course belongs to, the 'Semester' the course is taken in, the 'Department' which teaches the course ("CSC" is the Computer Science department) and the 'Year' in which the course should be taken. Major nodes contain their 'MajorID' (e.g. "CSC") and are connected to course nodes by edges of the type `REQUIRES` (see Figure S4.1). `REQUIRES` edges contain a 'type' attribute which can either be "Compulsory" or "Alternate". Edges of type "Alternate" contain an additional 'Combination' attribute used to group alternate equivalent courses together to ensure that pairs of courses are taken together. For example, in the case where, either MAM1000W is to be taken alone or MAM1004F and MAM1008S are to be taken together, the combination value stored in the `REQUIRES` edge between the major node is 1 for both MAM1004F and MAM1008S and 2 for MAM1000W. Using this, if MAM1004F is present in the student's planned curriculum, checking that the other required course (MAM1008S) is also taken involves finding all courses of combination 1.

See Figure S4.2 for all the marks achieved by Student 2591978 in Computer Science courses and Highschool subjects. Figure S4.2 also shows the relationships between the Computer Science major node and the courses it requires.

5.2 Constraint Checking Graph Design

The decisions required to check the constraints of a degree are represented in a Constraint-Checking graph (Figure S5.2), like that implemented by De Marzi [21]. Consisting of Question and Terminal nodes, this graph receives a list of course counts and NQF totals (Table S2.2) and checks that each adheres to their limits specified in rules FB 3 and 7 [26].

Question nodes only contain a 'Name' attribute which describes the rule it represents and a 'Code' attribute which stores the corresponding rule code (e.g. "FB7.7(a)"). Terminal nodes only contain a 'Name': "Meets the requirements" or "Does not meet the requirements". Each Question node is connected to at most one other Question node and at least one Terminal node (Figure S5.2). Nodes are connected by `DECISION` edges containing an 'answer', 'description', 'direction', 'parameter' and 'value' (see Figure S5.1).

Each count/ total value passed to the graph is evaluated in the `DECISION` relationships and not the nodes themselves. The 'answer' attributes (either "Yes" or "No") are used to improve the understandability of the graph and are directly related to the 'description' attributes which elaborate on the answer. The 'direction' attribute (either "Left" or "Right") is used to direct the traversal to the correct Terminal node. The 'parameter' attribute specifies which count is to be checked and the 'value' attribute corresponds to the limit of that count.

The count to be checked can either be less than the 'value' attribute or greater/ equal to the value. Whenever the count is less than the value then the 'direction' attribute is always "Left." Likewise, if the count is greater than or equal to the value then the 'direction' attribute is "Right." If the count illegally exceeds the value (a "Right" direction) then the relationship connects the question node to the "Does Not Meet requirements" Terminal node. And if the count is legally less than the value of the relationship (a "Left" direction) then the relationship connects the Question node to the next Question node (provided it is not the final Question node).

Figure S5.1 shows the two `DECISION` edges connected to the first Question node. This node corresponds to checking that the curriculum contains the equivalent of 9 full courses. Hence the parameter to be passed to both `DECISION` edges is "fullCourses", a count value obtained from the Student Data graph. The 'value' parameter is 9, facilitating the check that fullCourses > 9.

Further details regarding the traversal through the Constraint Graph are found in Section 5.3.2.

5.3 Queries

Two types of queries are required to extract the necessary data from the Student Data graph. These queries are contained within a Java wrapper program which facilitates user input, retrieval of query results, passing of data to the Constraint Checking graph and output to the user.

5.3.1 Curriculum Queries on the Student Graph

To obtain the data required to complete checks 1 to 4 outlined in Section 3, the following queries were developed. A few example queries have been selected to demonstrate concepts but have been shortened for clarity and do not represent the totality of queries required.

5.3.1.1 Find all courses required by a given major which are not in the planned curriculum

Firstly, to find all courses which are compulsory to a Computer Science Major, all nodes connected to the CSC major node by a `REQUIRES` edge of type "Compulsory" are collected. After passing in the student's planned curriculum as a string (line 3), all courses which are compulsory but do not appear in the planned curriculum are filtered in the square brackets on line 3. These are returned as `missingCourses` on line 4.

```
1.  MATCH (m: Major {MajorID: "CSC")-
    [:REQUIRES {type: "Compulsory"}]
            ->(c: Course) AS compCourses
2.  //pass in planned curriculum here
3.  [n in compCourses WHERE n
    NOT IN curriculum] AS missingCourses
4.  RETURN missingCourses
```

In a similar manner as above, all alternate equivalent courses in a major are collected. Then using the 'combination' attribute, any alternate missing courses are returned.

5.3.1.2 *Find the counts and total NQF credits of specific course types*

Firstly, all course nodes listed in the student's curriculum are collected. From these course nodes, multiple different counts can be obtained by filtering and summing. Roughly 30 individual count and total values (Table S2.2) are required to generate the values passed to the Constraint Checking graph (Figure S5.2). A single `MATCH` statement was devised to retrieve all of these counts at once, as the system aimed to minimise database reads for performance.

To count all half year senior science courses taken in the curriculum, the code snippet below is used to filter the node results of the `MATCH` (line 1). Due to the complexities of filtering and summing various node attributes, the `SUM` and `SIZE` functions need to be used together.

```
1.  MATCH (c: Course) AS courses
2.  WHERE c.CourseName IN [ curriculum ]
3.  RETURN SUM(SIZE([course IN courses
            WHERE course.Faculty = "SCI"
            AND (course.Semester = 1
            OR course.Semester = 2)
            AND (course.Year = 2
            OR course.Year = 3)]))
            AS seniorSciHalfCourses
```

To retrieve the total NQF credits for certain types of courses, all the nodes in the student's curriculum are first collected as above. In the case of the sciNQF (the total NQF credits of all courses taken in the Science Faculty), the following steps are required:

1. All non-science course nodes are filtered out of the student's curriculum in the square brackets.
2. The 'Credits' attribute is extracted from each course node in the filtered list using `EXTRACT` and `node.Credits`.
3. `REDUCE`, `total` and `accumulator` are used to sum the 'Credits' attribute of each node with 0, to obtain the total Credit value of all the filtered courses.

```
4.  RETURN REDUCE (accumulator = 0,
            total IN EXTRACT(node IN
            [course IN courses
            WHERE course.Faculty = "SCI"]
            | node.Credits)
            | accumulator + total) AS sciNQF
```

These procedures are repeated with different criteria to collect all the count values listed in Table S2.2. Once all the values have been collected, a dictionary of values is generated and passed to the Constraint Checking graph.

5.3.2 *Queries on the Constraint Checking Graph*

An example dictionary of count and total values could resemble:

```
{countFullCourses: 9, totalNQFCredits: 434, …}
```

The dictionary of counts is passed to the Constraint Checking graph as "counts" in line 1. In line 2 all possible paths (`p`) through the graph which start at the first Question node and end at a Terminal node are collected. Lines 3 to 5 filter the paths by the relationships (`r`) between the nodes. If `counts[r.parameter]<r.value`. then the "left" relationship is chosen (line 4) and if greater than/ equal to the value, then the "right" relationship is chosen (line 5).

Only one path exists which contains a chain of unique relationships defined by the input counts. The 'description' attribute of each relationship in the path is extracted in line 6. The name of the Terminal node and list of relationship descriptions return a logical path to the Terminal node in line 7.

```
1.  WITH counts
2.  MATCH p = (q:Question {start:1})-
    [d:Decision*]->(t:Terminal)
3.  WHERE  (r in relationships(p) WHERE
4.         (r.direction='left' AND
           counts[r.parameter]<r.value) OR
5.         (r.direction='right' AND
           counts[r.parameter]>=r.value)
6.  EXTRACT(r IN relationships(p)
    | r.description) AS extractedDescriptions
7.  RETURN t.name, extractedDescriptions
```

Consider the example of checking the first question node "9 Full courses?" (Figure S5.1) with the `fullCourses` parameter to be checked. The count stored in `counts[fullCourses]` is accessed and compared to the relationship value (`r.value`) of 9. Because the count of 9 >= the 'value' 9, the "right" direction is chosen. The "right" relationship connects the first question node to the second question node "6 Science Full Courses?" (see Figure S5.2). The relationships of this next question node are then evaluated, and the process is repeated until a Terminal node is reached.

5.3.3 *Similarity Queries on the Student Graph*

To generate an indication of similar student performance, the following queries are required.

5.3.3.1 *Find students with similar curricula*

Firstly, all student nodes which are connected to any courses in the student's curriculum by `ENROLLED_IN` relationships are found (line 1, 2). Next, the ID of the student is set as the item to be compared for similarity and the `courseID` is set as the category on which to compare the students (line 3). `data` is a dictionary containing an item (`studentID`) and a category (the list of courseIDs the student enrolled in). A `data` dictionary is created for each student found in the `MATCH` statement.

The Neo4j Jaccard similarity algorithm is used to compare each student in `data` and their enrolled courses to every other student in `data`. A call to `stream` parallelises the execution (line 4). All students who are not enrolled in at least 50% of the same courses are excluded in line 5 and returned in line 6.

```
1.  MATCH (s:Student)-[:ENROLLED_IN]
          ->(c:Course)
2.  WHERE c.CourseName IN [ curriculum ]
3.  WITH {item:id(s), categories:
          collect(id(c))} as data
4.  CALL algo.similarity.jaccard.stream(data)
    YIELD intersection
5.  WHERE intersection > minSimilarCourses
6.  RETURN data.asNode.StudentID
    AS similarCourseStudents
```

5.3.3.2 *Find students with similar NBT marks*

In the case of first years, all students connected to the NBT subject nodes and who are enrolled in similar courses (`similarCourseStudents` created by the above query) are

collected (line 1, 2). Then, as above, a `data` dictionary containing the `studentID` (as the item to compare) and marks achieved in each NBT exam (as the categories to compare each student) is created for each student (line 3). This is then passed to the Neo4j Euclidean distance similarity algorithm (line 4). Students with a Euclidean distance greater than 18 are not considered similar (as explained in Section 4.4.3) and are excluded using the `similarityCutoff`.

```
1.  MATCH (s:Student)-[r:ENROLLED_IN]
    ->(sub:Subject {"NBT"})
2.  WHERE s.StudentID IN[similarCourseStudents]
3.  WITH {item:id(s), weights:
    (coalesce(r.Mark))} as data
4.  CALL algo.similarity.euclidean.stream(
      data {similarityCutoff: 18})
5.  RETURN data.asNode.StudentID
      AS similarMarkStudents
```

The similarity categories (the student's marks) are padded with `coalesce` in line 3 so all arrays are of the same size. This process is similar for senior students and comparing their GPAs.

5.3.3.3 Find students with similar GPAs

When comparing single GPAs among senior students, the Euclidean Distance algorithm is equivalent to the absolute difference between marks (Section 4.4.3). Therefore, the algorithm can be implemented manually as follows:

```
1.  WHERE s.StudentID IN[similarMarkStudents]
2.  WITH [s.StudentID,(avg(r.Mark))] AS map
3.  [n in map where n[1]>LB and n[1]<UB] AS
    filteredStudents
4.  RETURN filteredStudents[0]
```

After creating a map of a studentID to their GPA, each student is filtered according to their GPA lying in a range between LB and UB, 5% below and above the user's input mark respectively (line 3). The corresponding StudenIDs are extracted in line 4.

5.3.3.4 Find the predicted performance of the student

To provide a prediction of the user's performance next year, the group of students who have now been selected according to their similar curricula and marks are used. The average GPA (and accompanying standard deviation) of this group of students in the next year is returned as a predicted GPA (e.g. if the student is entering first-year, then the average first-year GPA of all students with similar NBT marks is returned).

```
1.  MATCH (s:Student)-[r:ENROLLED_IN]
          ->(:Course {Year: nextYear} )
2.  WHERE s.StudentID IN [filteredStudents]
3.  RETURN (avg(r.Mark)), (stDev(r.Mark))
```

6 SQL IMPLEMENTATION

The same small data set inserted in the Neo4j V1 graph in Section 5 was then inserted into a MySQL relational database.

6.1 MySQL Database Design

The relational representation of the data stored in MySQL (see Figure S6.1) stored marks in one large table. The Students table contains a list of Student IDs, the Subject table contained a list of all Subject IDs and Subject Names (including NBT 'subjects'). The Courses table contained a list of Course IDs, with corresponding Course Names, NQF credits, the faculty, department and year the course is taken in. The Requires table stores all courses which are required by a major (stored in the Majors table) in addition to the course combinations, to replicate the `REQUIRES` edge design in Section 5.1 and process in Section 5.3.1.1. Instead of including the attributes "Compulsory" and "Alternate", the 'Combination' value is used. A course with a Combination of 0 is a core compulsory course, and all other positive integers encode various combinations of alternate equivalent courses.

The SubjectMarks table and CourseMarks table represent student marks achieved in subjects taken before UCT and courses taken at UCT respectively. Each record is of the form: `StudentID, CourseID, Mark` representing the mark attained by a specific student in a specific course.

6.2 Queries

The following queries are required to facilitate the constraint checking and grade prediction. A wrapper program is used to facilitate input and output, but unlike the Neo4j implementation, the degree rules are not stored in the database. A 'constraint-checking table' is not a typical structure and course counts are not unnecessarily passed to a database query. Instead, the constraints of the degree are checked through if-statements (equivalent to the Question nodes) within the Java program.

6.2.1 Curriculum Queries

To retrieve the curriculum course count values, a `SELECT COUNT` query counts the number of courses which apply to a criterion by checking "curriculum" variable passed in on line 3. The `SUM` function is used instead of `COUNT` to retrieve NQF credit totals.

```
1.  SELECT count(CourseID) AS sciHalfCourses
2.  FROM courses
3.  WHERE courseName IN (curriculum)
4.  AND (Semester = 1 OR Semester = 2)
5.  AND Faculty = "SCI"
```

To find compulsory courses missing from the student's curriculum, a JOIN between the Requires and Courses table is needed (line 3). The tables are JOINed on CourseID and those courses which have a Combination of 0 (and are compulsory) and are not in the curriculum are selected.

```
1.  SELECT c.CourseName as compCourses
2.  FROM courses c
3.  right JOIN requires r on
          r.CourseID = c.CourseID
4.  AND Combination = 0
5.  AND MajorID = "CSC"
6.  WHERE c.CourseName is not null
7.  AND c.CourseName not in (curriculum)
```

To select alternate equivalent courses not taken in a complete combination, a similar query as above is required with the condition that the course Combination does not equal 0.

6.2.2 Similarity Queries

Firstly, students with similar courses are selected. This requires a JOIN between the Courses and CourseMarks tables (line 2) to compare the course names – CourseMarks stores all the CourseIDs a student has taken, but the CourseName is required for

comparison. The query finds all students who attained a mark in any of the courses in the student's input curriculum (line 3) and counts the number of times each StudentID appears. If the count of a StudentID (grouped on line 4) is greater than minCourses (50% of the students planned curriculum on line 5) then that student enrolled in at least half the same courses as the user.

```
1.  SELECT StudentID AS similarCourseStudents
    FROM coursemarks cm
2.  right JOIN courses c
          on c.CourseID = cm.CourseID
3.  WHERE c.CourseName in (curriculum)
4.  GROUP BY StudentID
5.  HAVING count(StudentID) > minCourses
```

Finding students with similar NBT marks does not require a JOIN as the SubjectIDs for NBTs are known to be 4,5 and 6.

Those students with similar curricula are filtered according their marks lying between LB and UB, 5% below and above the user's average input mark respectively. Average marks achieved in the student's current year (specified in line 4) are used for comparing senior students' GPA (line 6), whereas individual NBT marks are compared for first-year students.

```
1.  SELECT StudentID AS similarMarkStudents
2.  FROM coursemarks cm
3.  WHERE StudentID in(similarCourseStudents)
4.  AND cm.Year = 2
5.  GROUP BY StudentID
6.  HAVING avg(Mark) > LB and avg(Mark)< UB
```

The overall average mark achieved by similarMarkStudents (the result of the previous query) in the following year is then retrieved as predicted performance.

```
1.  SELECT avg(Mark), stddev(Mark)
2.  FROM coursemarks cm
3.  WHERE StudentID in (similarMarkStudents)
4.  AND cm.Year = 3
```

7 NEO4J V2 & MYSQL V2 DESIGN

To accurately compare the performance of Neo4j and MySQL, much larger databases were required. Massive volumes of student data were not readily available and had to be generated. With larger data sets, new relationships were introduced in the graph representation to ensure performance benefits.

MySQL's schema required no changes and as such MySQL V2 used the same design and queries described in Section 6 with the larger data set.

7.1 Data Set Generation

Not enough existing student data was available, so data was generated to represent UCT students as realistically as possible. The simulated students were enrolled in pseudo-correct curricula, meaning the courses enrolled in, and the marks achieved in those courses where not totally randomised. The purpose of this simulated data was performance testing and not the accuracy of predictions produced.

Students (and their marks) were generated using a Monte Carlo simulation as follows. The probability tables used to simulate students can be seen in Tables S7.1, S7.2 and S7.3 and all random numbers were generated in the range [0,1) unless stated otherwise.

Two random numbers, F1 and F2, were generated to assign a student to two faculties (e.g. Commerce and Science). It is common for a student to enrol in the courses of one faculty only; to represent this, a third random number, α, was generated. If α was less than 0.4, then both of the student's majors were taken from F1, else, both F1 and F2 were used to assign majors. The faculties were assigned to F1 and F2 using the proportions in Table S7.1, calculated from the original, smaller data set.

Next, two new random numbers were generated, M1 and M2, to assign a student a major within each faculty. No data was available on the number of students who take specific majors, so M1 and M2 were used to pick majors at random within the student's faculties (e.g. Economics and Computer Science). As students taken between 3 and 9 courses from the department of their major, 2 random numbers, C1 and C2, in the range [3,9] were used to create curricula. So, if C1 and C2 were 6 and 9, then 6 courses from M1 and 9 courses from M2 would be selected as the student's curriculum.

To ensure reasonable cluster sizes for students with identical curricula, entire groups of students of size K were assigned the same curriculum. This would allow for reliable mark predictions as the grades predicted would be calculated from larger clusters of size K. K was randomly selected between [50, 320] to represent the average similar student cluster size of 185 which existed in the smaller data set. The curriculum of each student was hashed to produce `CurriculumHash`, used to group students with the same curriculum.

Using the distribution of marks of all students of the smaller dataset in Table S7.2 and S7.3, two random numbers, GPA_R and NBT_R, were used to assign a student a range of marks for their courses and NBTs respectively. If GPA_R corresponded to the range [50,60) (see Table S7.2) then all marks for all courses were assigned marks between 50% and 59% at random.

1 million students, consisting of 5441 clusters of students with identical curricula were generated.

7.2 Neo4j V2 Design

The following design changes to Neo4j V1 were made to reduce the search space during queries and improve performance.

Student nodes in Neo4j V1 (Section 5.1) were never directly connected but were indirectly connected through `ENROLLED_IN` edges connected to a common Course node. To decrease the number of lookups required during queries, a tree structure was implemented (Figure S7.4). This tree structure introduced direct relationships between students which would reduce the number of nodes to be queried when using the similarity algorithms.

From Figure S7.4, a new node type StudentRoot and two new relationships `TOOK` and `CHILD_OF` were added to the design of Neo4j V1. StudentRoot nodes are the only type of student node connected to Major nodes through `TOOK` relationships as they represent an entire cluster of students with identical curricula. Once a student's curriculum is hashed, if a StudentRoot with the corresponding `CurriculumHash` does not exist, that student becomes a StudentRoot, else the student is connected to the StudentRoot node with the corresponding `CurriculumHash` through a `CHILD_OF` relationship. The attributes of nodes and relationships remained the same as in Neo4j V1 with the addition of the `CurriculumHash` attribute in Student(Root) nodes.

StudentRoots have K Student node children as described in Section 7.1. See Figure S7.5 for the majors taken and courses enrolled in by StudentRoot 285121, and Figure S7.6 for all the student nodes which have the same `CurriculumHash` and are children of StudentRoot 285121.

7.3 Neo4j V2 Queries

With the new node and relationship types, the queries in Section 5.3 were optimised to improve performance on the larger dataset. When finding similar students, all StudentRoot nodes connected to the student's input majors are found. As 5441 clusters exist in the data, the node search space is immediately reduced from 1 million to around 5 thousand. Typically, the number of StudentRoots connected to any two majors is between 1 and 10. The student's input curriculum is first hashed to determine if any StudentRoot shares the same curriculum. If not, the Jaccard similarity algorithm is used to determine which StudentRoot has the curriculum most like that of the user, where a maximum of 10 nodes are likely to be compared. This number of nodes compared using Jaccard is much less than with Neo4j V1, where hundreds of students could be compared for similarity.

Next, all the marks of all the students connected to the most similar StudentRoot node through `CHILD_OF` relationships are found and grade prediction continues as described in Section 5.3.3.2 onward.

8 EXPERIMENT DESIGN

To compare the performance of each database implementation, query retrieval times were measured for the queries described in Section 5.3, 6.2 and 7.3. All experiments were run on databases installed on an HP laptop running Windows 10 x64 with an i7-7500U CPU @ 2.70GHz, 2 Cores and 8 GB of installed physical memory (RAM).

Neo4j and MySQL V1 databases each contained 2776 students, 25061 marks achieved across 434 courses and 4387 marks achieved across 7 subjects. Neo4j and MySQL V2 databases each contained 1 million students, 11,932,548 marks achieved across 2396 courses and 3,000,000 marks achieved in 3 NBT subjects.

All Neo4j graphs, including the Constraint Checking graph containing 14 nodes and 24 relationships, ran on version 3.5.8 Enterprise of Neo4j. Neo4j V1 was allocated a maximum Heap of 1 GB, and 4 GB for Neo4j V2. The MySQL databases ran MySQL Community server version 8.0.17.

8.1 Query Retrieval Times

Each of the following tests were run 3 times each in Neo4j and MySQL and the query retrieval times were measured for the 3 different runs on a warm start. No other applications were running during the execution of each query. Tests 1 and 2 timed the course count queries (Section 5.3.1.2. and 6.2.1) and were provided with the same Computer Science curriculum plan.

1. Single MATCH vs Single SELECT

To compare the performance of the same types of database reads, the same query written in Neo4j and SQL were compared. The time taken for a MATCH statement (with one return variable) to retrieve the same count value as the SELECT equivalent were recorded for all of the queries which calculate the values listed in Table S2.2. This test determines which database can execute read operations faster.

2. Single MATCH vs Multiple SELECT

Multiple different count/ total variables are filtered from the nodes collected in a single MATCH statement. SELECT statements do not allow for all (30+) values to be collected from one SELECT. Using the Java currentTimeMillis() method this test compared the time taken for the single MATCH query to execute and return the results and the total time taken by all separate SELECT queries.

3. Constraint Checking graph Traversal vs if-Statements

The average time taken for traversal of the Constraint Checking graph to return was compared to the total execution time of all if-statements implemented in Java. Three input curricula were tested: one which violates no constraints, one which violates the first constraint and one which violates the last constraint. This test determined if representing decision logic in Neo4j has any performance advantages.

4. Neo4j Similarity Algorithms vs MySQL SELECT

The processes involved in finding similar students (finding students with similar courses, then similar marks, then predicting their GPA) are the most computationally expensive and thus completion time was compared in both data sets (V1 and V2 for both Neo4j and MySQL). The similarity processes were timed for three different students with different majors, year of study and GPA (see Table S8.1).

9 EXPERIMENT RESULTS

Table 2: Average Neo4j & MySQL query Completion times

Query	Average Time (s)
Single MATCH with single return	0.002
Single SELECT single return	0.0005
Single MATCH with multiple return	0.015
Multiple SELECT with single return	0.017

Table 3: Average Graph Traversal & if-Statements Completion times

	Neo4j Traversal (s)	if-Statement (s)
Time to 1st violation	0.042	< 0.001

Table 4: Average Similarity Query Completion time

Query	Average Time (s)			
	Neo4j V1	MySQL V1	Neo4j V2	MySQL V2
1. Similar Course Students (Jaccard)	5.575	0.053	0.037	0.311
2. Similar Mark Students (Euclidean Distance)	0.031	0.014	0.035	7.224
3. Mark Prediction	0.016	0.013	0.012	6.764

10 DISCUSSION OF RESULTS

10.1 General Query Performance

Table 2 reflects MySQL's performance advantage when accessing one variable with one database read. The average completion time for one Neo4j MATCH statement with one return variable is 2 milliseconds, whereas SQL retrieved the same value on average 4 times faster (0.5 milliseconds). Whilst both database reads were fast, the overheads involved in retrieving node objects likely caused the slower reads compared to SQL's more advanced disk reads. It is more likely that the rows and tables were stored contiguously on

disk and less likely that node objects were stored together. Also, as theorised in [23], the conversion between Strings and Integers in Neo4j may have impacted performance. Therefore, the retrieval of induvial counts in MySQL was significantly faster than in Neo4j.

However, on average, Neo4j's ability to run 1 query (1 database read) to retrieve multiple counts in 15 ms was clearly advantageous when compared to the sum of MySQL's individual queries (multiple reads) which ran in 17 ms. Whilst the performance of both types of query are similar, Neo4j has a statistically significant performance advantage; a two-sample t-test confirmed that the null hypothesis that the average completion time of a single MATCH is equal to the average completion time of multiple SELECTs is false. Although the time difference between the queries was on average 2 milliseconds, this test showed that despite very fast reads of a database, a single slower query in Neo4j has a performance advantage over many fast queries in MySQL.

From Table 3, the traversal of a constraint-checking graph was not as efficient as a chain of if-statements. The time taken for the Neo4j Constraint Checking graph to traverse to a Terminal node was not slow but was simply not as fast as Java if-statements. Although representing decision logic in a graph has visual benefits, the complexities involved in creating such a graph and query traversals are unjustified when, for a brief list of checks, it was simpler and faster to implement as if-statements. The benefit, however, of the schema-less nature of Neo4j is that if any changes are made to the constraints of a degree, very few modifications would need to be made to the graph; whereas reflecting changes in a complex chain of if-statements would require considerably more effort and lines of code.

10.2 Similarity Query Performance

Tables S8.1 and S8.2 summarise the input and output of each run of the similarity queries.

In Table 4, the queries implemented in MySQL V1 all execute faster than the Neo4j V1 equivalent. The performance differences are greatest in the first query when finding similar course students. Neo4j takes multiple seconds to execute whilst MySQL completes in milliseconds. All Neo4j similarity algorithms are designed to compare each node input to every other node in the input list; the number of student nodes compared is $(number\ of\ students\ input)^2$ - this has a time complexity of $O(N^2)$ because each student is compared to every other student input into the algorithm [16]. In MySQL, similar students are found by updating counts whilst iterating through a table – a time complexity of O(N). The cause of the massive time difference for the first query is Neo4j V1 computing many more comparisons than MySQL V1. For student 1 in Table S8.2, over 2 thousand students enrolled in at least one of their listed courses, this results in over 4 million comparisons made in Neo4j but just 2 thousand in MySQL. The time difference is less noticeable for queries 2 and 3 because the number of students compared is much smaller (180 comparisons for student 1 in query 2 in Neo4j and 19 for MySQL). The performance differences of query 3 are less noticeable as the final lookup of grades is typically within a small group of students and.

Neo4j V2 clearly outperformed MySQL V2. The use of the StudentRoot and `CHILD_OF` relationships to form a tree structure and reduce the node search space lead to massive speedup. The fact that Neo4j V2 contained a graph approximately 4 million times larger than Neo4j V1, and still performed 150 times faster for query 1 justifies the addition of more relationships to create a more structured graph. The JOINs required in MySQL V2 are significantly greater and this had a large impact on performance. Query 1 was 10 times slower in MySQL V2 than Neo4j V2 and more than 200 times slower in queries 2 and 3. Queries 2 and 3 involved more operations on the columns JOINed (see Section 6.2.2) and thus performed the worst.

11 CONCLUSIONS

11.1 Neo4j allows database reads to be minimised.

Cypher syntax allows for many variables to be returned from one MATCH statement. This means that the database only needs to be read once, and calculations can be made thereafter. This is not possible in SQL; specific columns need to be declared in SELECT statements and results cannot be filtered after the fact to produce different results. This has performance benefits as one database read with post-processing is faster than multiple database reads without post-processing.

11.2 MySQL is more efficient in smaller data sets.

Every query tested executed faster in MySQL for the smaller data used in V1. The more advanced lookups and fewer overheads involved in searching records as opposed to nodes lead to faster performance in every aspect. The smaller data set required no large JOINs and as such, there were no performance advantages of graph traversals over JOINs.

11.3 Identifying similar students is more efficient in Neo4j in large databases.

The comparison between the larger databases showed that the performance degradation when JOINing massive tables impacted the speed of identifying similar students. Because of the tree structure implemented in Neo4j V2, the number of student nodes compared was minimised as only the necessary edges between relevant nodes were traversed. In MySQL V2, millions of records were JOINed and therefore performance suffered. In cases where many thousands of data points need to be evaluated and compared, Neo4j is likely to outperform relational tables.

11.4 Storing constraints in a database is too costly.

For a simple list of checks, the complexity involved in developing a constraint checking graph is not justified by fast performance; a chain of if-statements is better suited. However, more complex decision logic (like classification trees etc) is perhaps more suited to graph representation because of the visual benefits of graphs. Although advantageous when making changes to the requirements of the constraint-checking graph, schema independence provided no performance improvements in this context.

12 LIMITATIONS

The largest limitation of this research was the small data set of real data that system was built on. The 50% lower bound course intersection was too broad to provide accurate predictions but was necessary to create clusters greater than 0 in the small data set. This limit was too lenient but necessary for the system to be functional when used with the smaller database.

13 FUTURE WORK

Beyond developing a user interface, the main proposed improvements to be made concern the grade prediction. With a large enough data set, experimentation with the 50% 5% course and mark similarity cut-offs can be conducted to identify similar students more accurately. Further, student similarity determination could consider: the marks achieved in specific courses, taken in a specific year, in a certain course load as other factors.

ACKNOWLEDGEMENTS

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

REFERENCES

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

[2] 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.

[3] Chabane, S. et al. 2018. *Occupations in high demand in South Africa*.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] Neo4j Experimental Similarity algorithms: *https://neo4j.com/docs/graph-algorithms/current/experimental-algorithms/similarity/*. Accessed: 2019-08-31.

[16] Neo4j Jaccard Similarity algorithm - Similarity algorithms: *https://neo4j.com/docs/graph-algorithms/current/experimental-algorithms/jaccard/*. Accessed: 2019-09-01.

[17] Neo4j Memory Configuration: *https://neo4j.com/docs/operations-manual/current/performance/memory-configuration/*. Accessed: 2019-08-31.

[18] Neo4j Tuning of the garbage collector: *https://neo4j.com/docs/operations-manual/current/performance/gc-tuning/*. Accessed: 2019-05-01.

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

[20] 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.

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

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

[23] 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.

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

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

[26] 2019. University of Cape Town Faculty of Science Handbook.

SUPPLEMENTARY IMFORMATION

S1: Graph Database Design

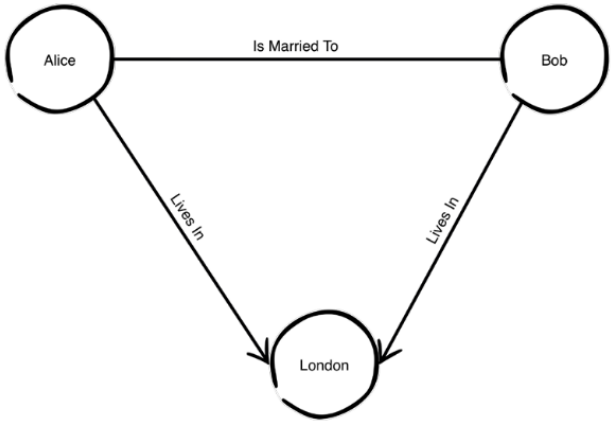

Figure S1: A graph representation of data [9]

S2: System Design

S2.1: UCT Science handbook rules

- FB3 A student shall not register for more than: (a) the equivalent of four half-courses in each semester in the first academic year of study; (b) the equivalent of three half-courses in each semester in any other year of study.
- FB7.1 The curriculum shall include the equivalent of at least nine full-year courses of which at least six full-year courses must be Science courses.
- FB7.2 The curriculum shall include the equivalent of at least four full-year senior courses or the equivalent, of which at least three shall be Science courses, and the equivalent of two full-year courses shall be third-year (level 7) courses.
- FB7.7 The standard BSc degree requires: (a) a total of 420 NQF credits (nine full-year courses). A minimum of 396 NQF credits will be accepted where the second major or suite of hierarchical courses includes at least one senior full course from another Faculty(b) a minimum of 276 NQF credits from Science courses (the equivalent of six full-year courses) (c) a minimum of 120 NQF credits at level 7

S2.2 System Variables

Table S2.2: Variables to be extracted from the databases

Variable	Description
S_iY_j	The number of courses taken in semester i of year j. $i \in \{0,1,2\}, j \in \{1,2,3\}$
`seniorHalfCourses, seniorFullCourses`	The total number of Half/ Full senior courses.
`halfCourses, fullCourses`	The total number of Half/ Full courses taken.
`sciHalfCourses, sciFullCourses`	The total number of Half/ Full science courses.
`seniorSciHalfCourses, seniorSciFullCourses`	The total number of Half/ Full senior science courses.
`thirdHalfCourses, thirdFullCourses`	The total number of Half/ Full taken in year 3.
`totalNQF`	The total NQF credits of all courses in the curriculum.
`sciNQF`	The total NQF credits of all Science courses .
`thirdNQF`	The total NQF credits of all third-year courses.
S_iY_jNQF	The total NQF credits of all courses in semester i of year j. $i \in \{0,1,2\}, j \in \{1,2,3\}$

S3: System Processes

S3.1 System Flowchart

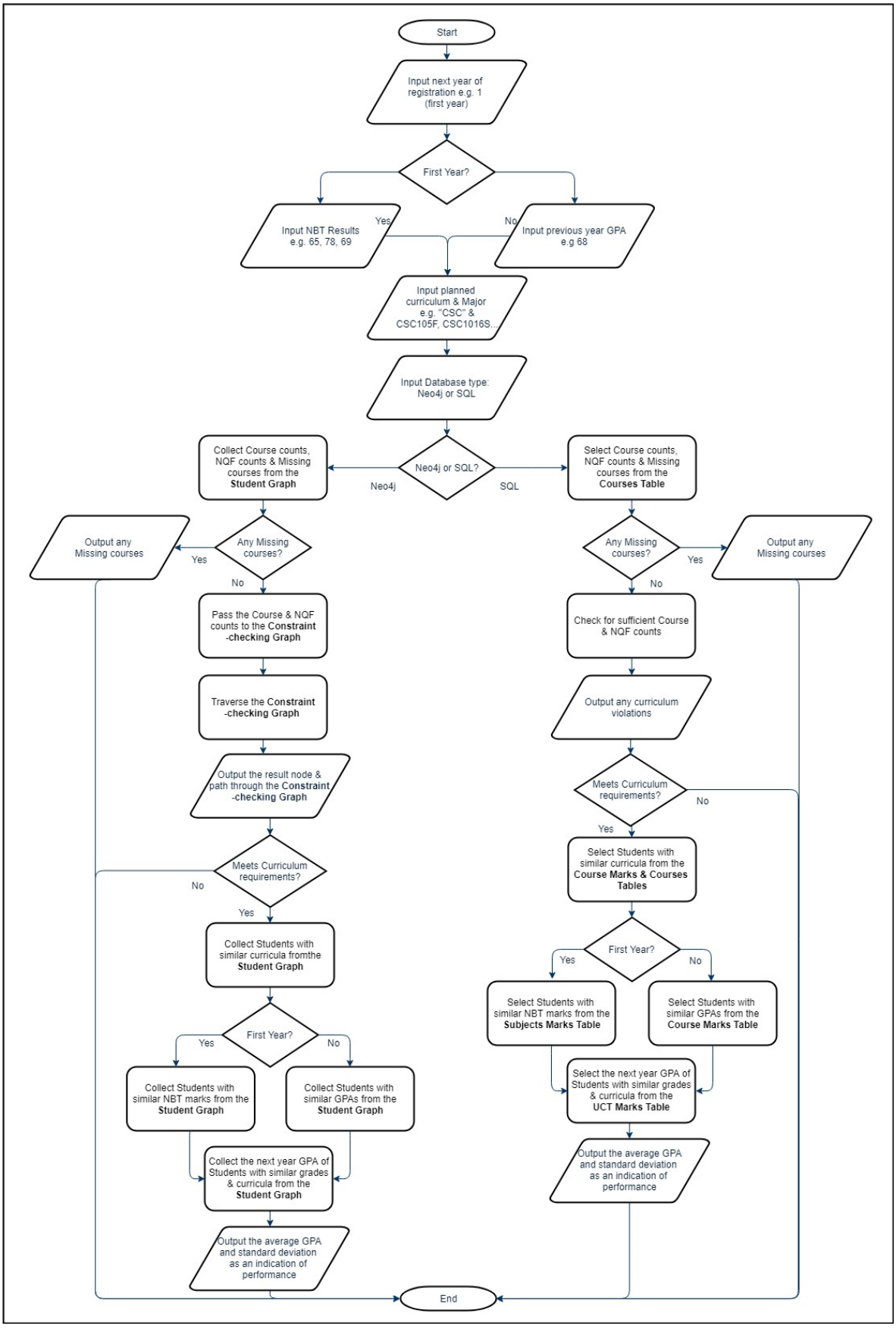

Figure S3.1: System Flowchart

S3.2 System Input & Output

```
Enter the which year you are registering for: 1, 2 or 3
>1
Enter your NBT Academic Literacy, NBT Math and NBT Quantitative Literacy marks separated by commas
e.g. 65, 75, 50
>65, 75, 71
Enter your curriculum Major(s) e.g. CSC, STA
>CSC, STA
Enter your entire curriculum (list of all courses from year 1 to 3) e.g. CSC1015F, MAM1000W...
>ACC1006F, MAM1000W, CSC1015F, ECO1010F, STA1006S, CSC1016S, ECO1011S, CSC2001F, INF2009F,
INF2006F, STA2020F, STA2030S, CSC2002S, CSC2003S, CSC3002F, STA3030F, CSC3003S, STA3036S
Enter which database you want to use: Neo4j or MySQL
>MySQL

Retrieving compulsory courses...
Checking compulsory course constraints...

+------------------------------------------------------------+
| Curriculum meets all the course requirements of the Major. |
+------------------------------------------------------------+

Retrieving course counts...
Checking course count constraints...

+-----------------------------------+
| Enough Full Courses               |
| Enough Science Full Courses       |
| Enough Senior Full Courses        |
| Enough Senior Science Full Courses |
| Enough Third Year Full Courses    |
| Enough First Year Half Courses    |
| Enough Senior Year Half Courses   |
| Enough NQF Credits                |
| Enough Science NQF Credits        |
| Enough Third Year NQF Credits     |
+-----------------------------------+

+-----------------------------------+
| Curriculum Meets Count Requirements |
+-----------------------------------+
Finding similar students for grade prediction...
+-------------------------------------------------------------------------+
| Found 52 past students who enrolled in similar courses to your curriculum. |
+-------------------------------------------------------------------------+
+-----------------------------------------------------------------------------------------------------+
| Of 52 students enrolled in similar courses, 12 students achieved a NBT mark similar to you (70).|
+-----------------------------------------------------------------------------------------------------+
+-----------------------------------------------------------------------------------------------------+
| The average first year GPA of 12 students with a NBT and curriculum similar to you is 63 with a |
| standard deviation of 11.                                                                           |
+-----------------------------------------------------------------------------------------------------+
```

S4: Neo4j V1 Student Graph Design

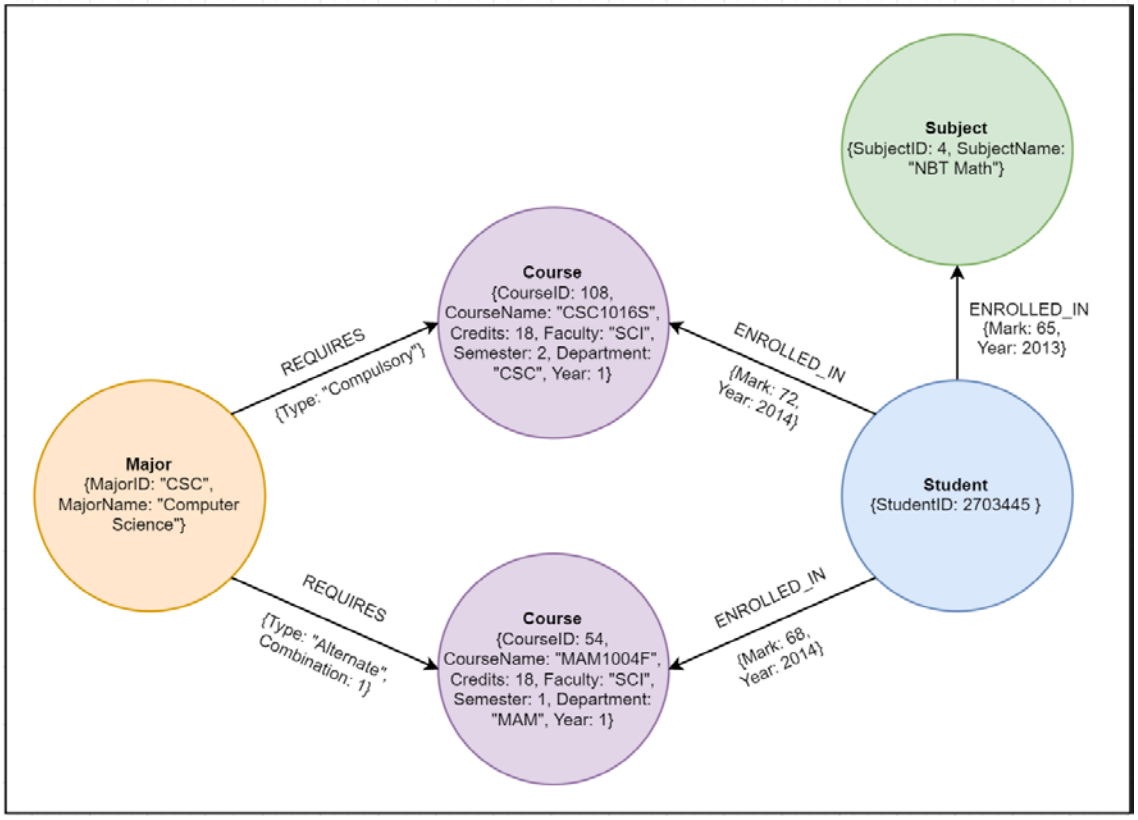

Figure S4.1: Student Data Graph Design

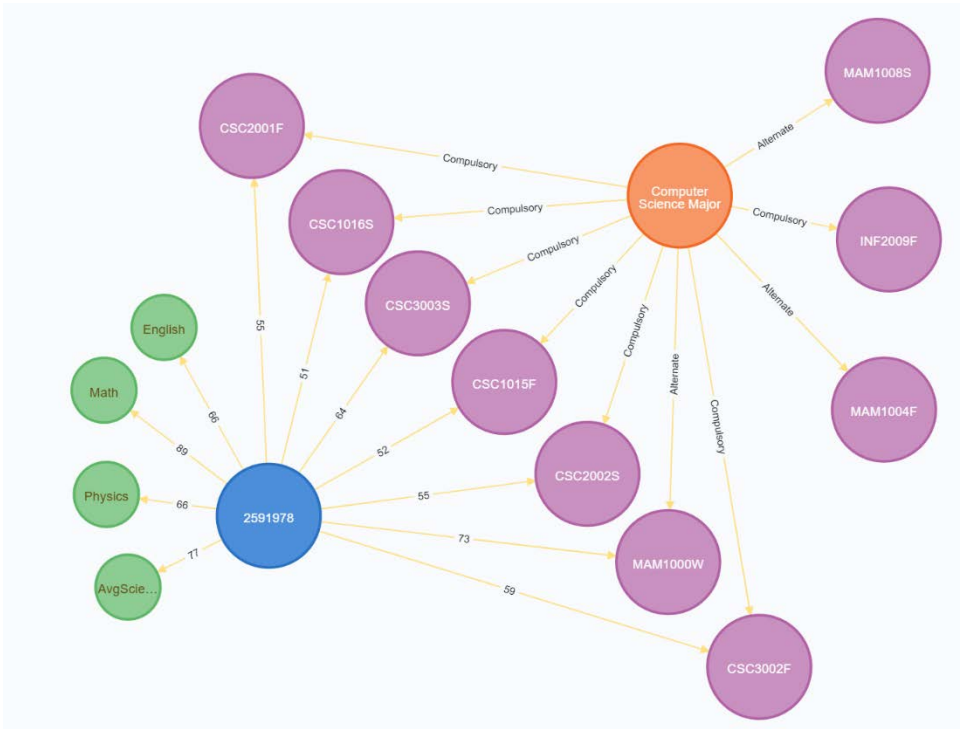

Figure S4.2: Marks achieved by Student 2591978 & Courses required by a Computer Science Major

S5: Constraint Checking Graph

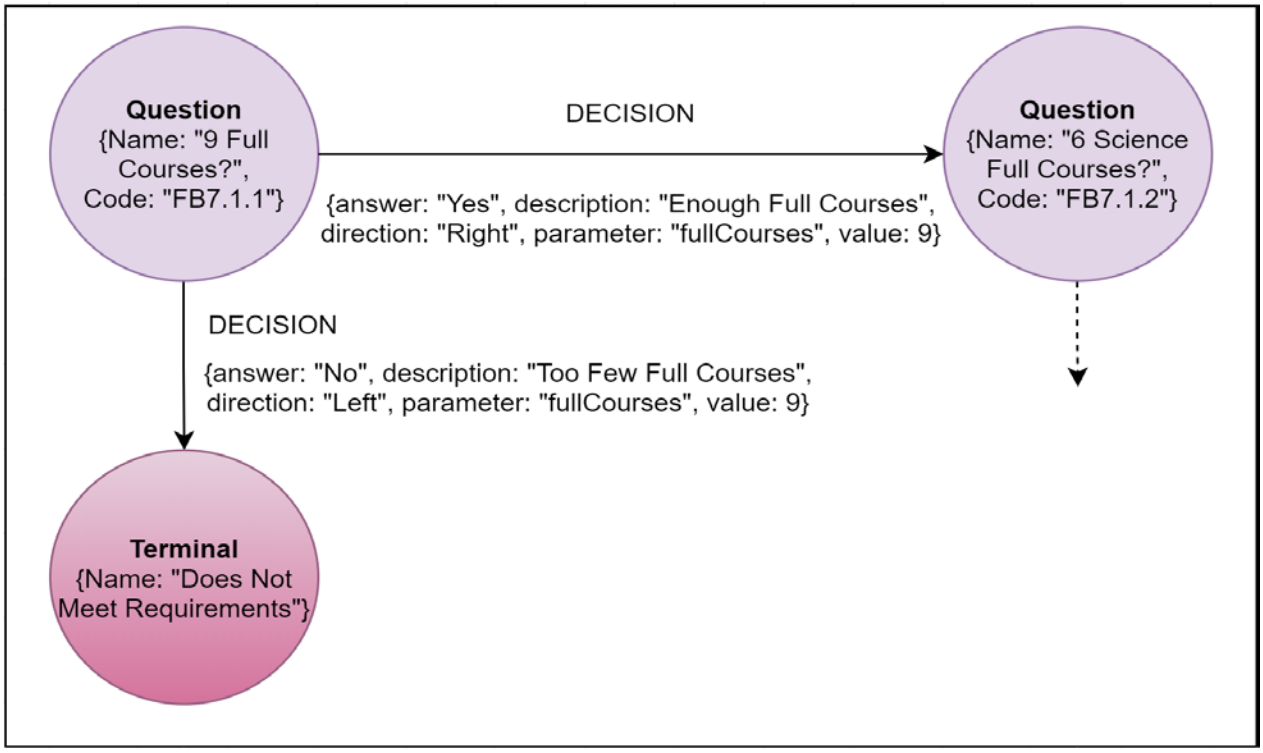

Figure S5.1: Constraint Checking Graph Design

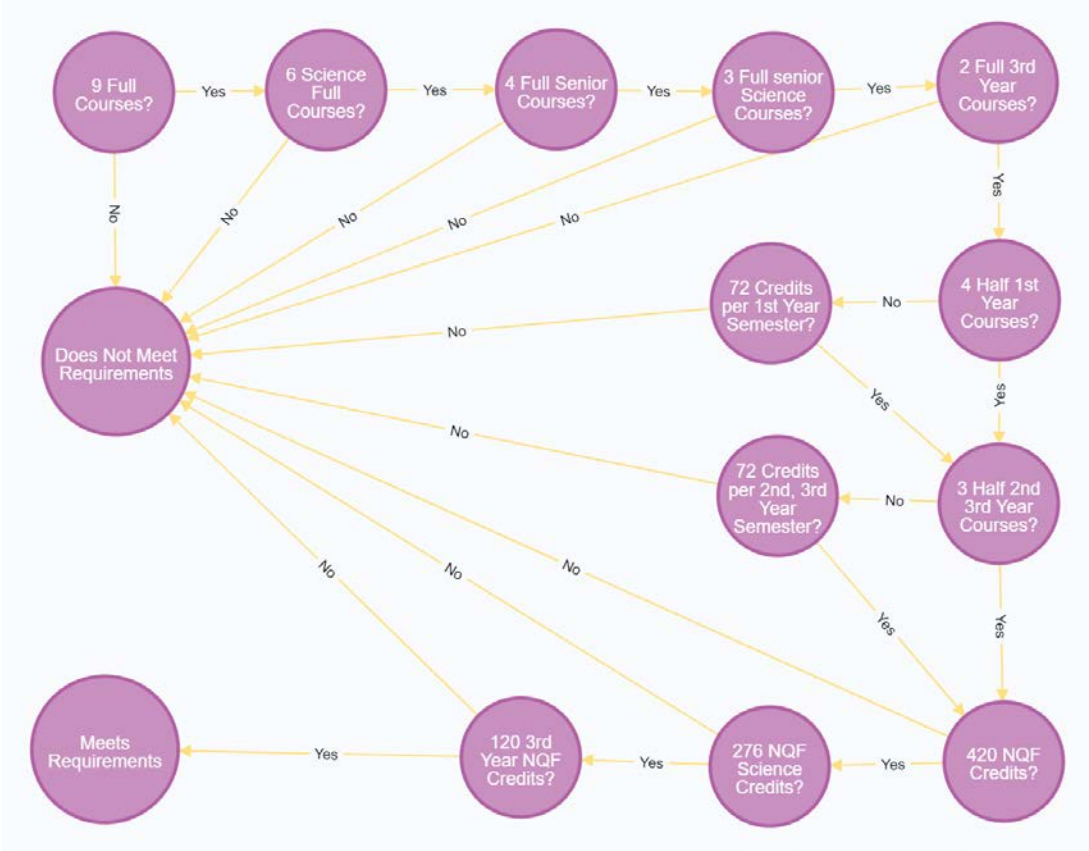

Figure S5.2: Neo4j Constraint Checking Graph

S6: MySQL Schema

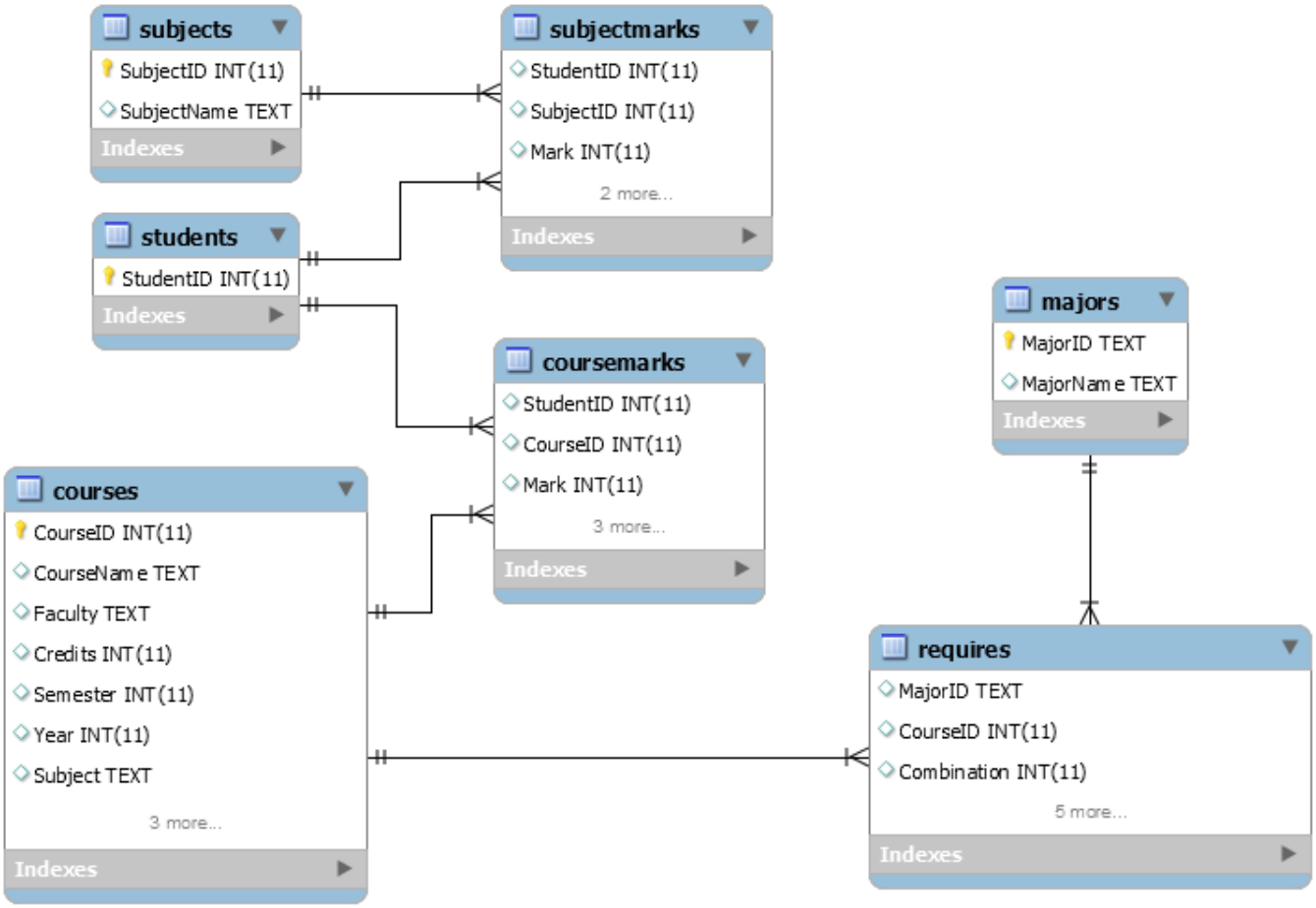

Figure S6.1: Relational Student Database Schema Design

S7: Neo4j V2 Design

Table S7.1: Faculty proportions

Faculty	Proportion	Cumulative Probability	Random Number Range
COM	0.311056	0.311056	$0 \leq F1, F2 < 0.31$
EBE	0.165122	0.476178	$0.31 \leq F1, F2 < 0.48$
HUM	0.301658	0.777836	$0.48 \leq F1, F2 < 0.78$
LAW	0.015925	0.793761	$0.78 \leq F1, F2 < 0.79$
MED	0.094505	0.888265	$0.79 \leq F1, F2 < 0.89$
SCI	0.111735	1	$0.89 \leq F1, F2 < 1$

Table S7.2: GPA Grade proportions

GPA Range	Proportion	Cumulative probability	Random Number Range
$0 < GPA < 50$	0.3450	0.34	$0 \leq GPA_R < 0.34$
$50 \leq GPA < 60$	0.2048	0.55	$0.34 \leq GPA_R < 0.55$
$60 \leq GPA < 70$	0.2923	0.84	$0.55 \leq GPA_R < 0.84$
$70 \leq GPA < 80$	0.1332	0.98	$0.84 \leq GPA_R < 0.98$
$80 \leq GPA < 100$	0.0239	1.00	$0.98 \leq GPA_R < 1$

Table S7.3: NBT Grade proportions

NBT Range	Proportion	Cumulative probability	Random Number Range
$0 < NBT \leq 50$	0.2247	0.22	$0 \leq NBT_R < 0.22$
$50 \leq NBT < 60$	0.1482	0.37	$0.22 \leq NBT_R < 0.37$
$60 \leq NBT < 70$	0.1988	0.57	$0.37 \leq NBT_R < 0.57$
$70 \leq NBT < 80$	0.2398	0.81	$0.57 \leq NBT_R < 0.81$
$80 \leq NBT < 90$	0.1520	0.96	$0.81 \leq NBT_R < 0.96$
$90 \leq NBT < 100$	0.0363	1.00	$0.96 \leq NBT_R < 1$

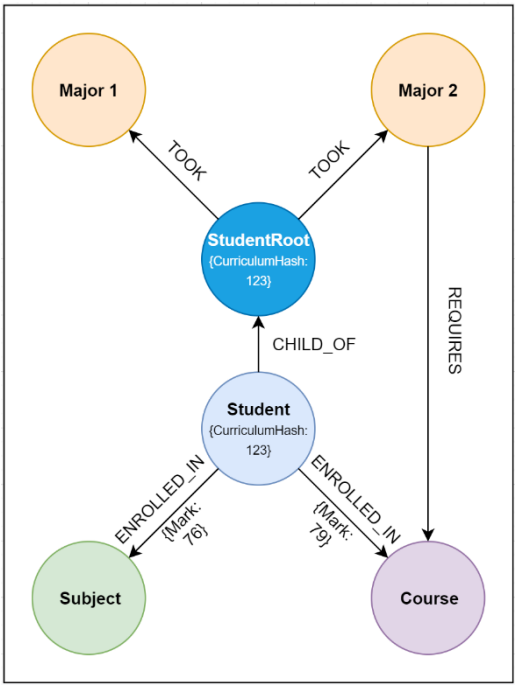

Figure S7.4: Neo4j V2 Student Graph Design

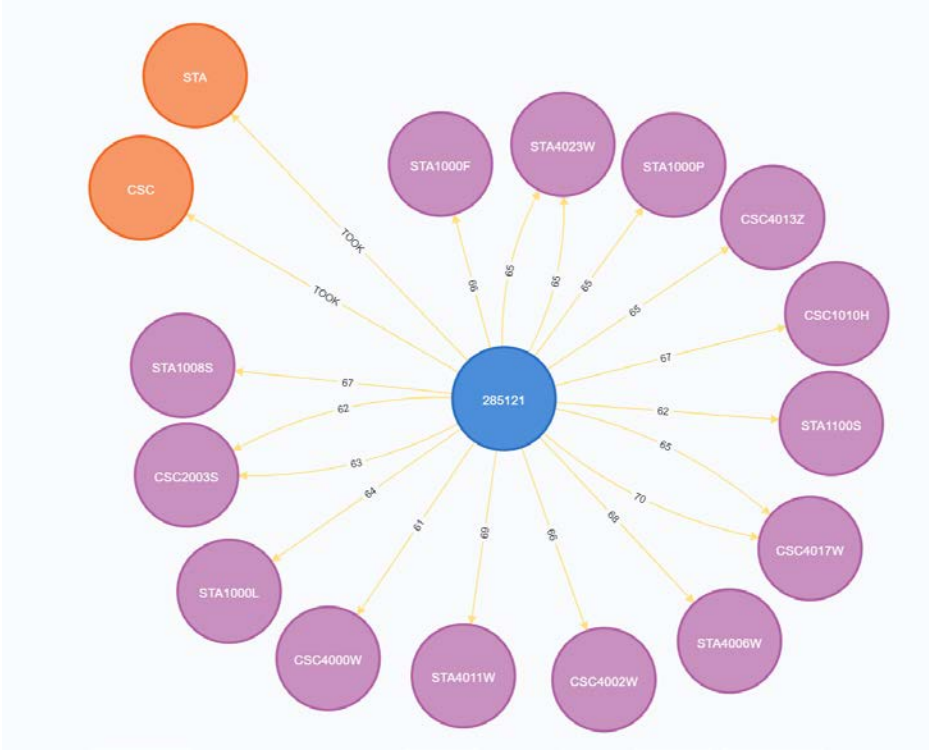

Figure S7.5: Majors taken, courses enrolled in and marks achieved by StudentRoot 285121

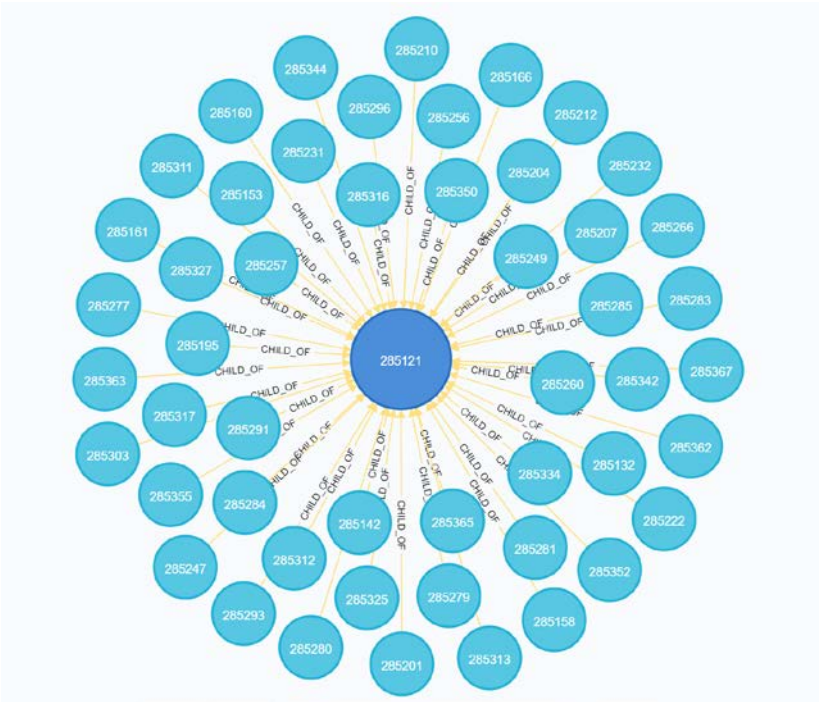

Figure S7.6: Student nodes which are children of StudentRoot 285121

StudentRoot 285121 represents all students with identical curricula who took Computer Science (CSC) and Statistics (STA) majors. All the children nodes in Figure S7.6 took all the same courses as StudentRoot 285121 took in Figure S7.5.

S8: Similarity Query input and output

Table S8.1: Students and input curricula used for GPA Prediction

Student	Majors	Courses	Input Year	Input Mark(s)
1	CSC BIO	MAM1000W, CSC3002F, CSC3003S, CSC1015F, CSC1016S, INF2009F, CSC2002S, CSC2001F, CSC2003S, BIO1004S, CEM1000W, MCB2020F, MCB2021F, MCB2022S, STA1007S, MCB3024S, MCB3025F, BIO1004F	1	[61, 72, 74]
2	CSC MAM	CSC1015F, CSC1016S, INF2009F, CSC2001F, CSC2002S, CSC2003S, CSC3002F, CSC3003S, MAM1000W, MAM1019H, MAM2000W, MAM3000W	2	65
3	CSC STA	CSC1015F, CSC1016S, INF2009F, CSC2001F, CSC2002S, CSC2003S, CSC3002F, CSC3003S, MAM1000W, STA1006S, SAT2004F, STA2020S, STA3041F, STA3043S	3	81

Table S8.2: GPA Prediction results for Neo4j & MySQL

Student	Number of students who took at least 1 course from the curriculum	Number of similar course students	Number of similar mark students	GPA	ST Dev
1	2132	19	5	65	11
2	1379	335	108	64	9
3	1316	146	23	75	14

Table S8.1 contains the data (their planned year of study, input curriculum and corresponding marks) describing 3 students used as input for GPA prediction, where the major column represents the list of courses input.

Table S8.2 contains the output GPA prediction results (with accompanying standard deviation) for each student in both Neo4j and MySQL V1 implementations, where ‘Number of similar Course students’ is the size of the groups of students found by the queries which are similar according to input courses and ‘Number of similar mark students’ is the number of students within the similar course student groups which have similar marks.