Neo4j: The Advising Matrix

Casey Walters
InspirEd Lab
OMIS 695

“There’s a difference between knowing the path and walking the path.”

- Morpheus

Table of Contents

Introduction

:Technology:

Databases are organized collections of data. There are lots of different ways to organize this data. Graph databases focus on the relationships between the data rather than the data itself. A graph database is organized in a network. Data is represented in nodes, properties, tags, and edges. Advantages of graph databases are performance, flexibility, and agility. Graph databases are simple and have fast retrieval. They are scalable and adapt to new business requirements much easier than databases using the relational model. The database schema can be altered, where a relational database would need to follow the schema it was built using. Neo4j.com lists 5 areas graph databases are being used: fraud detection, real time recommendation engines, master data management, network/IT operations, and identity and access management.

Graph databases do not have an official query language, but many use SPARQL, like Amazon Neptune, IBM DB2, and Allegro. Neo4j uses Cypher, which "is a declarative, SQL-inspired language for describing visual patterns in graphs using ASCII-Art syntax" (Cypher Query Language Developer Guides & Tutorials, n.d.).

Current attitudes in business seem to align with the idea that one should choose a database based off the data model, rather than try to fit the data into a relational model. Recently, graph databases have seen a surge in popularity because they can handle some of the complex questions easier than a relational database would.

For example, say you want to know the name of that one romantic comedy that had the actor/writer and the actor that played the lead in "The Matrix".

With the graph model, you would first locate the "The Matrix" movie node, as seen in figure 1.0.

You would then follow the "lead" edge to find the node "Keanu Reeves", the actor who played the lead in The Matrix.

You would find all the movies Keanu Reeves has also acted in, and then filter those movies with the category "romantic comedies".

Then you would take those results and write a query that looks for a movie that had an actor lead *and* write in the movie.

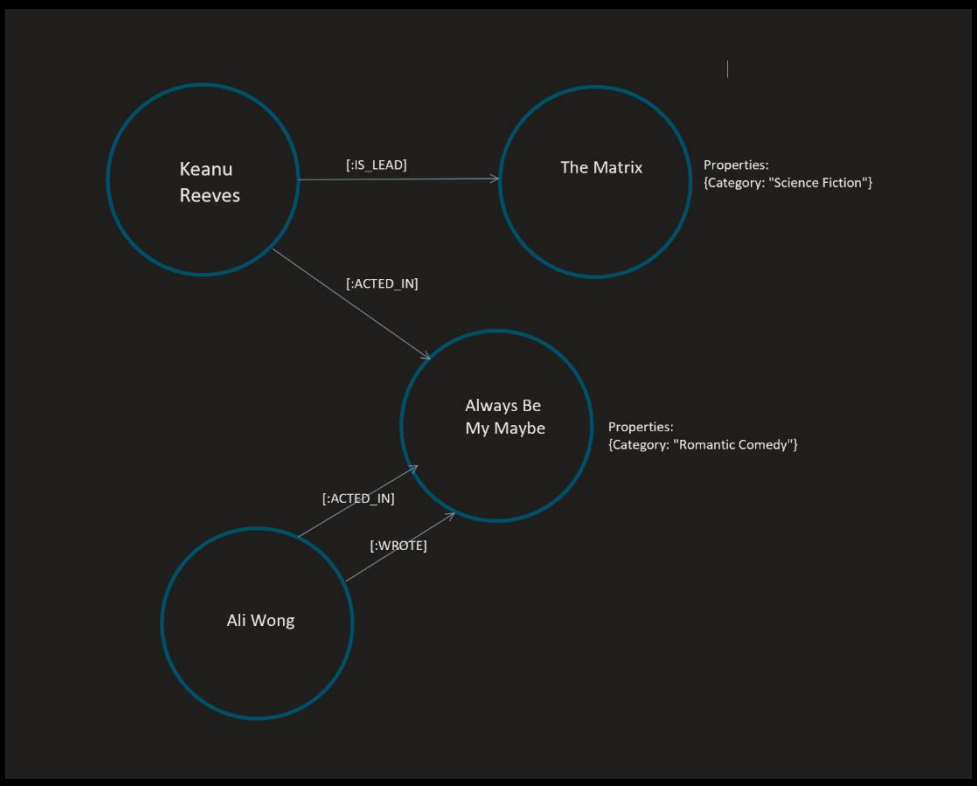

Figure 1.0: The graph model simplifies some queries that would get very complex in a relational database using SQL.

This would involve plenty of complex queries in a relational model but is rather simple with the graph model. However, there are plenty of situations where a SQL database makes more sense for the business.

:Business_Problem:

Currently, the OMIS Academic Advisor meets face to face with dozens of students a day, discussing their schedules for next semester. There are a fixed number of classes; and only so many unique combinations of courses being offered, enrolled in, and counting towards specific degrees. There is no true data stored on the courses at NIU, even a CSV file with the course names, numbers, and descriptions. Instead, we were told to use the course catalog.

Without a logical way to store this data, we are forced to rely on human brain power to perform these complex queries quickly and accurately every time.

:Proposed_Business_Solution:

By creating a graph database to represent the relationships between the courses in the OMIS department, we are providing a faster, more accurate system for students to use when trying to select their classes. This also removes stress on the academic advisor. By having students use this system for easy course scheduling questions, we free up time the academic advisor can now use to do other things in their job description.

Method

:Neo4j:

Neo4j is a popular graph database used internationally. The first ever version of Neo4J was developed in 2002, and the Swedish-based company's creation followed in 2007 ("Company", n.d.). Neo4j is "one of the most popular property graph databases that stores graphs natively on disk and provides a framework for traversing graphs and executing graph operations" (Francis et al., 2015).

Neo4j is a unique offering amongst other graph databases. *History of Databases and Graph Database* claims "Neo4j's storage engine uses fixed-size arrays to store the graph data, and can search nodes and relationships in O(1) time. This can be achieved by using array structures not by indexes. And Neo4j insists that Neo4j is a native graph database and the others are not, because the other systems use other storage engines" (Admin, 2017). Neo4j is incredibly fast and simplifies complex business problems. This is why Neo4j was chosen as the graph database in this project.

:Cypher:

Cypher is a query language for graph databases. Cypher allows us to create nodes and relationships. It also allows us to assign properties and tags to these nodes. Please see figure 1.1 below.

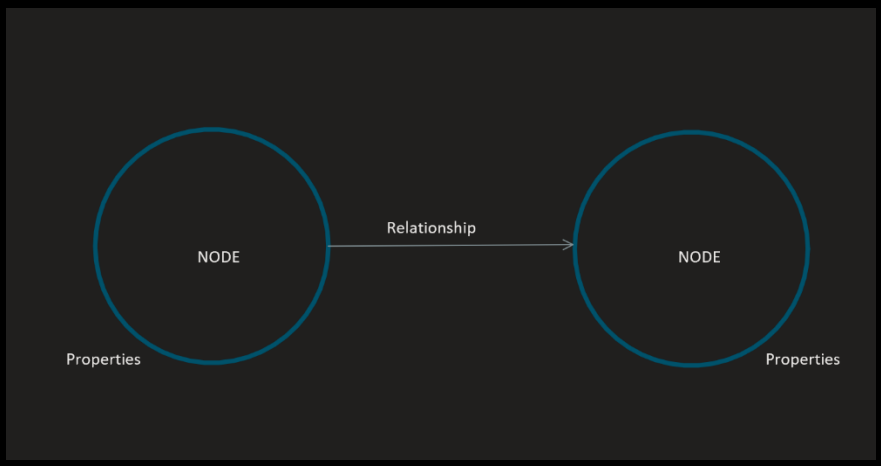

Figure 1.1: Cypher allows for creating these types of relationships. Nodes represent data, but data can also be stored within relationships. This is an advantage of graph databases.

Cypher borrows a lot of structure from SQL. For example, both SQL and Cypher use the WHERE and ORDER BY clauses. Cypher was created based on the graph model. It is a very visual language, and is based on ASCII Art. You will notice the use of parenthesis to create nodes, and the use of arrows to show relationships.

Cypher uses a few basic clauses:

- MATCH: Most common way to get your data from the graph.
- WHERE: Adds constraint to a pattern, or will filter the results of the query with the help of WITH.
- RETURN: What you want returned by the query. Similar functionality to the SELECT clause in SQL. (Chapter 1, n.d.)
- CREATE: Used to create a node in the database.

:Database_Schema:

Below is a sample of code used to make nodes. Use the CREATE clause to create a new node. I have specified that these nodes will be courses with the ":Course".

```
CREATE
(:Course {name: 'OMIS 352', desc: 'Managing Projects in Business'}),
(:Course {name: 'OMIS 449', desc: 'Business Application Development'}),
(:Course {name: 'OMIS 460', desc: 'Business Computing Environments and Networks'}
),
(:Course {name: 'OMIS 462', desc: 'Business Systems Analysis, Design, and Develop
ment'}),
(:Course {name: 'OMIS 475', desc: 'Internet and Web Computing Technologies'})
```

The name, which appears in the center of the node when you view the graph database, will always be the OMIS course number. I have assigned each course an attribute, named "desc" for description. This is where the true name of the class is stored. For example, the course OMIS 475: Internet and Web Computing Technologies is broken down into a node named OMIS 475, and it's description is "Internet and Web Computing Technologies".

Below is another example of creating a node.

```
CREATE (cert1:Certification { name: "IS Certification" })
```

This is the code used to create the IS Certification. I have used an alias here, cert1. The type of node is Certification, specified by the code ":Certification". I have also assigned an attribute, name. the name of the certification is "IS Certification".

```
MATCH (cert1:Certification { name: 'IS Certification' }),(p:Course { name: 'OMIS
352' })
MERGE (cert1)-[r:FULFILLED_BY]->(p)
RETURN cert1.name, type(r), p.name
```

Above is a sample of code that creates a relationship. I use MATCH to find two nodes; the IS Certification node and the OMIS 352 node. I want to create a relationship between these two nodes, so I use the MERGE clause. Notice the visual clarity of the code in this sentence. I specify that the IS Certification (which I nicknamed Cert1 back when I created the node) has a "fulfilled by" relationship with p. p is also a nickname, or alias, for a course. The course I selected in the MATCH clause was OMIS 352. This line states the IS Certification can be fulfilled by OMIS 352. Lastly, I wanted to return the name of cert1, the type of relationship between these two nodes, and the name of p, or the course.

Below is a snippet of code used to create the path "graduation". First, I used a MATCH statement to specify which nodes I was interested in creating a relationship between. I chose the nodes "First Semester Junior" and "Second Semester Junior", which are classified as "Standing" nodes. I have nicknamed them S1 and S2, respectively. I then used MERGE to create the relationship "Next Semester" on the path I named "Graduation".

```
MATCH (s1:Standing{name:"First Semester Junior"}),(s2:Standing { name: "Second Se
mester Junior" })
MERGE (s1)-[:NEXT_SEMESTER {path: "graduation"} ]->(s2)
```

This was done to create a progression within the database. By setting up the path 'First Semester Junior' to 'Second Semester Junior' to 'First Semester Senior' to 'Second Semester Senior' to 'Graduate', we are setting ourselves up to assign courses to different semesters. For example, a first semester junior would most likely be enrolled in the College of Business Core (OMIS 338, OMIS 351, etc). We create a relationship between these courses and the student standing.

Results

:Complex_Queries:

The most basic query discussed in this paper is figure 1.2 below. The cypher used to create this query is:

```
match (c) return c;
```

This code first runs to match any data related to "c", which is the alias for courses. Next, "return c" will return all the data relating to courses. This is the equivalent of a select all statement in SQL.

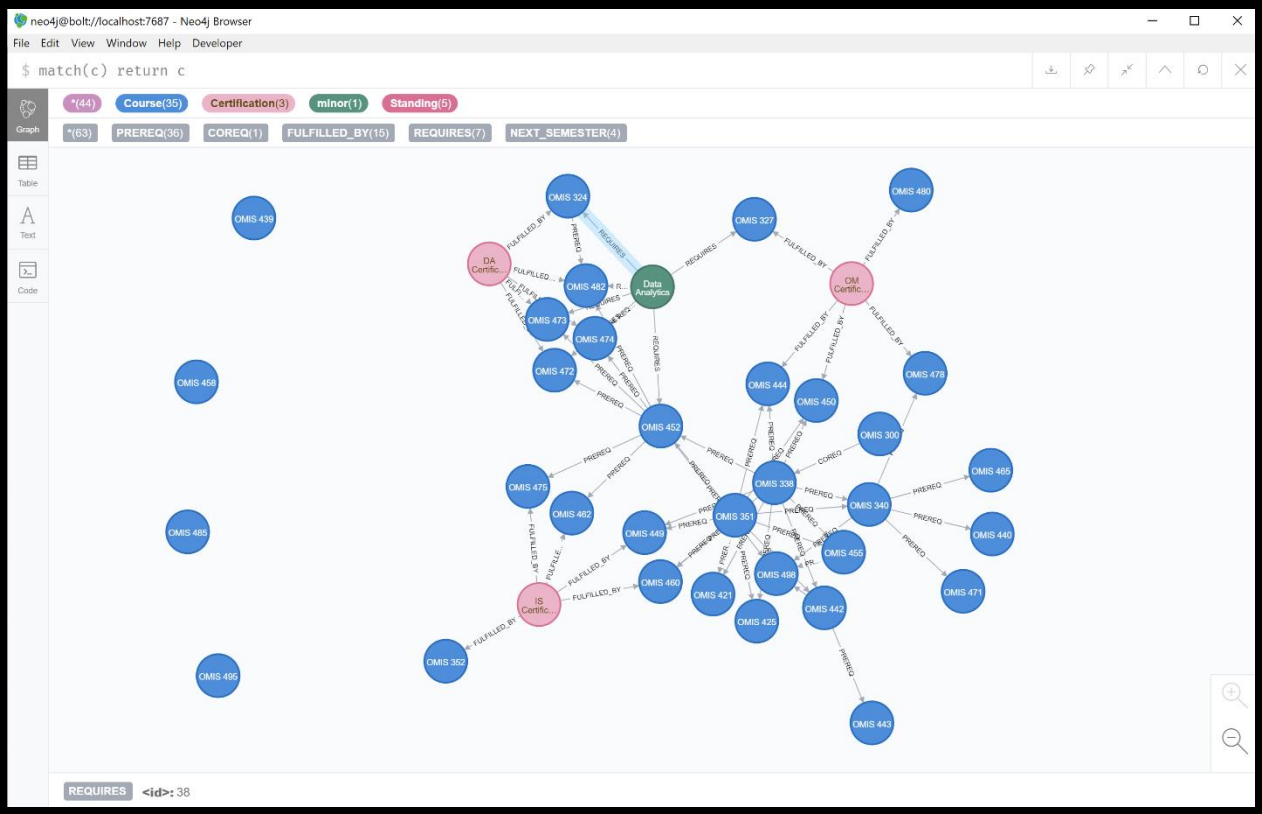

Figure 1.2: This is the contents of the entire advising database shown visually, and the output of a select all statement in Cypher.

Some courses in the department are listed as a prerequisite for other courses more than others. It would be beneficial for a student to take the courses that are more often listed as prerequisites earlier on than ones that are less often listed, in order to maximize the courses they can take in later semesters.

Say a student is enrolling in classes for their next semester. This student is particularly interested in expanding their options in future semesters. The student may very likely perform a query much like the one below to solve this issue.

```
match(e:Course{name: "OMIS 452"})-[:PREREQ]->(c:Course) Return c.name, c.desc;
```

This snippet of code is looking for the course named "OMIS 452". It will locate the OMIS 452 node and look for relationships named "PREREQ" that the OMIS 452 node has with other course nodes.

In English, this code is asking "which courses have the course OMIS 452 listed as a prerequisite?"

Now the student will be aware that if they take (and pass) OMIS 452, the courses returned from this query now become available for them to enroll next time around. (the courses returned may have other prerequisite relationships with other courses, so the student would ideally do this query a couple times with different classes, to see what would extend their opportunities most.) The results of the query are displayed in figure 1.3 below.

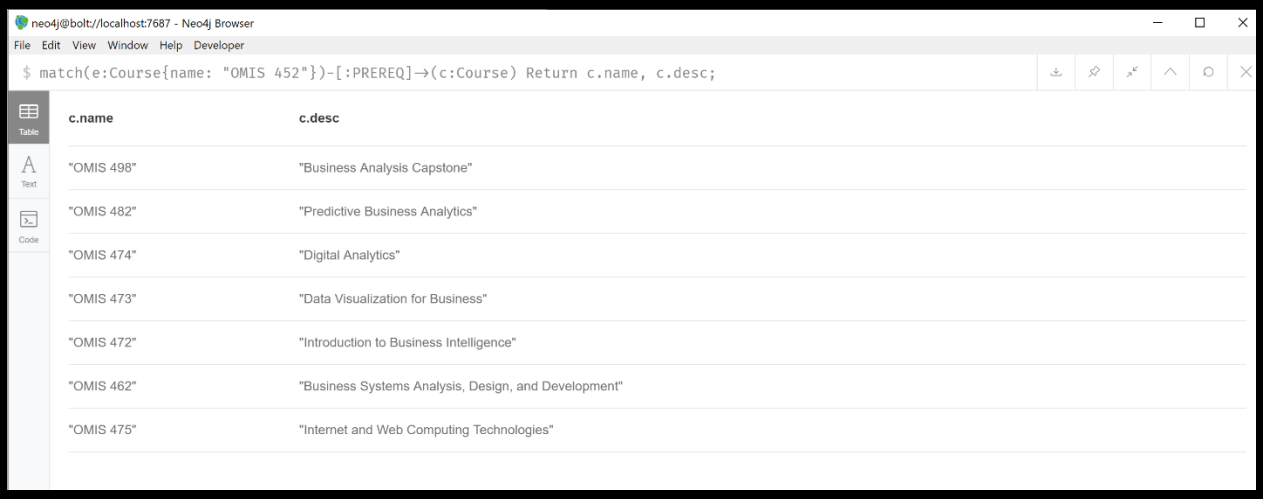

Figure 1.3: The results of the OMIS 452 prerequisite query.

Now let's continue with the same student in the previous example. The student can add on to the query to get more specific results. Say the student doesn't want OMIS 462 in the results. Maybe they are aware that OMIS 462 is taught online in the semester after this one, and they are not interested in taking the course for that reason. The student may specify they are uninterested receiving this course in the list of results using the following query.

```
match(e:Course{name: "OMIS 452"})-[:PREREQ]-
>(c:Course) where c.name <> "OMIS 462" Return c.name, c.desc;
```

This query is matching courses that have the relationship called "PREREQ" with the course named "OMIS 452". Return the course names and descriptions of courses that match this criterion AND that are not named "OMIS 462". See the results of this query in figure 1.4.

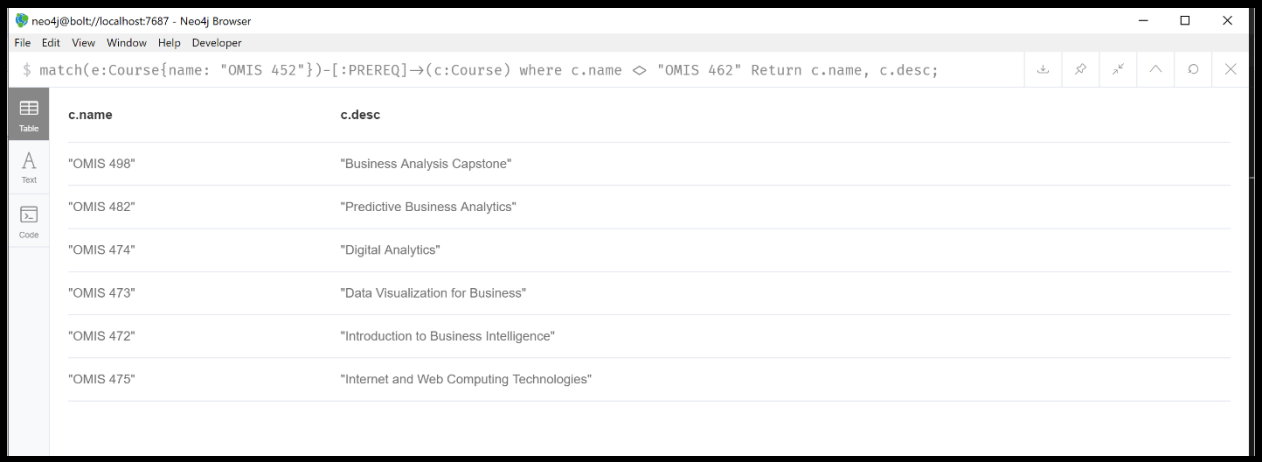

Figure 1.4: The results of the OMIS 452 prerequisite query without the OMIS 462 course being returned.

Say this student is interested in pursuing the Data Analytics certification. They may benefit from knowing which courses count for the Data Analytics certification that also require OMIS 452 as a prerequisite.

The Cypher below asks this question, and figure 1.5 displays the results.

```
match(e:Course{name: "OMIS 452"})-[:PREREQ]->(c:Course)<-[:FULFILLED_BY]-
(:Certification{name:"DA Certification"}) Return c.name, c.desc;
```

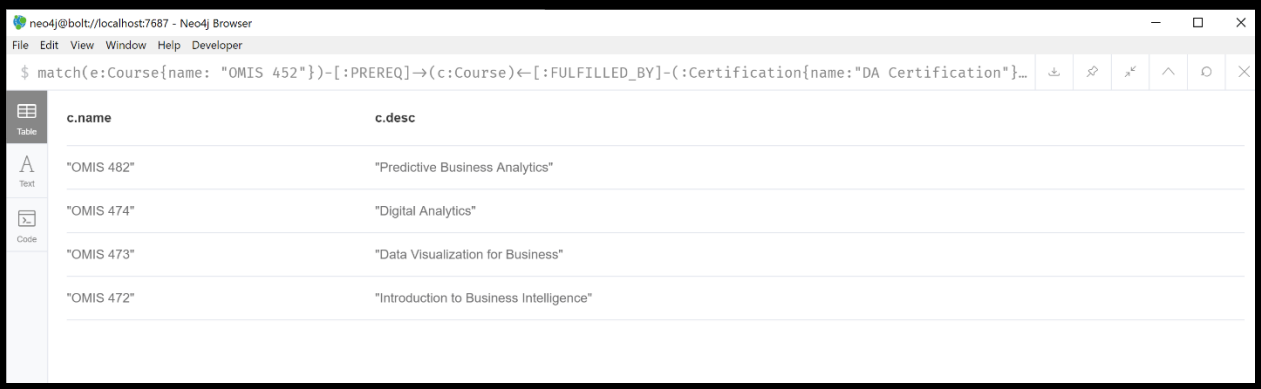

Figure 1.5: The courses returned all satisfy the following: a), the course is needed to fulfill the Data Analytics certification and b), the course has OMIS 452 as a prerequisite.

Now let's say a student is pursuing the IS certification and has room in their schedule next semester to take a class that will fulfill an IS certification requirement. However, this student has not taken OMIS 452 yet.

```
match (c:Course)<-[:FULFILLED_BY]-
(:Certification{name: "IS Certification"}) Where NOT  (c)-[:PREREQ]-
({name: "OMIS 452"}) Return c.name, c.desc;
```

This query is finding all the course nodes that have the "FULFILLED_BY" relationship type with the IS Certification node. However, this code is eliminating any of those courses from the results that have a "PREREQ" relationship with a course node named "OMIS 452". Figure 1.6 shows the results of this query.

Figure 1.6: The courses returned are the courses that will fulfill the IS certification but don't require taking OMIS 452 beforehand.

Discussion

:Next_Path:

Corequisites seem to be an interesting issue to represent graphically. Prerequisites are simple, because you must take one course before another. The Cypher needed to build this relationship is represented like this: (a)-[:is_prerequisite]->(b). However, a corequisite is used to express a course can be taken at the same time as another. Cypher requires relationships have a direction, but direction does not need to be specified in read queries. To get around this, I believe you can create the relationships both ways, and purposefully ignore direction while querying. For example, a and b are corequisites. Create the relationships (a)-[:IS_COREQ]->(b), (b)-[:IS_COREQ]->(a). Then when writing queries, don't specify a direction like this: (a)-[:IS_COREQ]-(b).

More properties should be added to course nodes. It would be beneficial for students to be able to use a WHERE statement to specify information such as instructors, professors who teach the course, when the course is offered(time, day of the week, semesterly), where the course is offered (NIU Naperville, main DeKalb campus, NIU Hoffman Estates), and possibly occupancy.

Learning paths can be used in collaboration with learning items and student standing. For example, students typically will take the College of Business core during their first semester of junior year. This could be paired with the suggested schedules online to provide students with more structure.

A web application is needed for this to become a viable business solution. While Cypher and Neo4j are easy to use, I don't see this database being adopted by students or faculty unless they can interact with some kind of application.

References

Admin. (2017, May 2). History of Databases and Graph Database. Retrieved from https://bitnine.net/blog-graph-database/history-of-databases-and-graph-database/

Chapter 1. Introduction. (n.d.). Retrieved April 29, 2020, from https://neo4j.com/docs/cypher-manual/current/introduction/#cypher-introduction

Company. (n.d.). Retrieved from https://neo4j.com/company/

Cypher Query Language Developer Guides & Tutorials. (n.d.). Retrieved from https://neo4j.com/developer/cypher-query-language/

Francis, N., Libkin, L., Plantikow, S., Green, A., Lindaaker, T., Rydberg, M., … Selmer, P. (2015, June 10). Cypher: An Evolving Query Language for Property Graphs. Retrieved April 29, 2020, from https://hal.archives-ouvertes.fr/hal-01803524/file/paper.pdf