Tanya Anisimova, Josh George, Brett Recker, Andrew Wilson

tatz8@mail.umkc.edu, jg6wy@mail.umkc.edu, bmrhdw@mail.umkc.edu, ajw664@mail.umkc.edu

A Comparison of Relational & Graph Database Efficiency

Submission Date: 12/6/2019

University of Missouri-Kansas City

Architecture of Database Management Systems (CS 5570)

Fall 2019

**ABSTRACT**

Relational and graph database models are detailed and compared. Using a dataset of users and their favorite books, outfitted for both models, queries are made to evaluate their performances. Results indicate that in a majority of circumstances, graph models are better suited queries that involve the retrieval of connected data.

**I. INTRODUCTION**

Growth of the global data sphere since the turn of the millennium has been tremendous; experts cite a doubling from 20 ZB to 40 ZB since 2016 alone, and predict a magnitude of 175 ZB by the year 2025 [1]. Of course, none of this data exists in isolation, but rather linked together in a myriad of ways. Online retailers may record a customer’s age, gender, location, purchase and browsing history, among other factors, in order to increase the precision of their advertising. Audio and video streaming services offer recommendations to their users based not simply on their past choices, but the choices of other users with similar tastes. Social media platforms present their users with networking suggestions based on mutual friends and shared interests [1]. It stands that the rich relationships between these innumerable pieces of data are just as important as the data itself.

What database structure both acknowledges and harnesses this parallel equality of information and connectivity? The solution is a graph database, a structure in which both data and relationships are equally first-class.

In this paper, we first outline the general architecture of relational and graph databases. Next, the performance of both models will be compared with regard to relationship-specific queries. We end with suggestions on the use of each model based on use cases and performance desires.

**II. BACKGROUND**

The relational database model is as follows: information is stored in tables, or relations, consisting of rows and columns. Each row in a table, called a tuple, represents an individual entity in the relation, while columns pertain to relevant attributes or categories of information. To store more complex collections of data, multiple tables may be utilized, assigning to each table a column of unique, non-duplicate values called primary keys. To connect tables to one another, tuples are given foreign keys that reference the primary keys of other tables [2].

Despite their name, relational databases deal poorly with data relationship queries beyond a depth of three levels, e.g. “find all users who are friends with another user that has family living in Hong Kong.” Producing results for these queries requires recursive join statements and can take much time, slowing down database performance.

In contrast, the graph database model gives equal priority to data nodes and their relationships. Nodes of data, labeled by their class (as in User, Product, Movie, etc.) may also contain attributes detailing their identity. Each relationship is an entity unto itself, labeled by its category (“Friend of”, “Employee of”, “Parent of”, etc.) and directed from one node to another. Relationships may be uni- or bidirectional [2].

One advantage inherent to graph databases is that locating pertinent information within them is analogous in practice to traversing a graph. Well documented in computer science, graphs are composed of nodes and edges, and are an essential data structure for storing information. Graph traversal and search algorithms are already thoroughly researched, eliminating the need for recursive and costly join queries.

Graph databases are versatile in their scalability. Inserting new nodes or relations into the database requires none of the refactoring that a relational database may require. Similarly, new categories of nodes or relations may be created on the fly without fear of altering the entire database. Similarly, the deletion of information from the database exists independently of all other data, with no risk of degrading existing information.

As of 2019, several graph database products are available. Neo4j, for which we found the most documentation and used in our experiment below, is utilized by dozens of companies, including Walmart, Lockheed Martin, and eBay, for managing their connected data [3]. Amazon Neptune, released for general use in 2018, is used by Samsung, Intuit, and Siemens [4]. Twitter developed its own FlockDB to optimize the reading and writing of their users’ follower graphs and tweet feeds [5]. Other graph databases available include ArangoDB, JanusGraph, and Cayley.

**III. EXISTING WORK**

Batra & Tyagi [6] assessed the differences between relational and graph models using MySQL and Neo4j. They collected experimental data by querying a dataset composed of users, their friends, their favorite movies, and the actors and actresses starring in those movies. The authors’ queries polled for information at increasing depths: finding all friends of a user, all favorite movies of a user’s friends, and all lead actors in a user’s friends’ favorite movies. They concluded that, in all cases of searching connected data, the graph database model produced query results faster than its relational counterpart, especially at increased dataset sizes.

**IV. CONCURRENCY CONTROL**

Most NoSQL databases have moved away from ACID compliance due to availability and performance, but Neo4j still provides support for these. This graph database model uses a read-committed isolation level. This means that only committed data can be read by a processing transaction. Data that has not yet been committed (still under process by an active transaction) is locked from other transactions. While this is weaker than serialization it does provide better performance. However, the Neo4j Java API provides locking capabilities at the node and relationship level which gives the effect of serialization (a higher isolation level). With this capability, when multiple write locks converge on the same node or relationship the transactions will serialize on the lock, limiting the possibility of processing conflicts. Within the Neo4j model, write locks occur during the addition, removal, or update of a node(s), a node’s property(ies), or relationship(s).

Since Neo4j uses a read-committed isolation level, non-repeatable and phantom reads may occur. A non-repeatable read is when data read two or more times by the same transaction is not ensured to be the same value. Since data is only locked for the duration of the read, transaction A could read a data item (x) first before transaction B updates this data item. Afterwards, if transaction A again reads data item x, the value will differ from the original read. A phantom read affects a transaction that performs a predicate selection multiple times, since it might observe a different result set each time, resulting in inconsistent behavior [7]. These are the two major concerns of Neo4j’s isolation level.

Adding and removing nodes or relationships in Neo4j can pose other issues as well, which is common across most database management systems. When deleting/removing a node, it first needs to be ensured that all attached relationships have been removed, as well as all the node’s properties. Before this removal has been committed, though, it is possible for a transaction to acquire a reference to that node. Neo4j does provide functionality to throw an exception if a transaction attempts to write to an uncommitted deleted node or relationship.

Additionally, put-if-absent functionality is available to avoid the creation of duplicate nodes by multiple concurrent threads. This is a race condition in which one of the concurrent transactions is deemed the winner and attempts to create the node (in turn, blocking the others). If the node creation is successful, the other transactions can process but will only return the details of the node entity that was just created. However, if the winning transaction fails to create the node, the race condition occurs again finding a new winner and repeating this process. Lastly, the TransactionTemplate class is also provided for handling deadlocks. This class automates a retry-on-exception loop. The number of retries and length of time to pause between retry can be configured using this class. These tools can be very helpful to avoid concurrency issues when integrating Neo4j into any system.

**V. COMPARISON**

The most common database models used in the industry today are relational, such as MySQL. These database models have some very stark differences when compared to graph models. For starters, when setting up a relational model with any system, denormalization is a necessity. Denormalization is the process of manipulating the user defined data model to suit the database engine, not the user. This must occur because normalized schemas aren’t fast enough for real-world use cases. Relational models can also become much more complex when compared to graph models due to the addition of foreign keys and join tables. This metadata is used to improve performance and reduce the number of joins, which is not necessary in graph models. An example of a join table could be implemented for a user’s email address. Usually this can be stored in a separate table, but since one user can have multiple emails it may be necessary to add two (or more) columns to an existing user table for quicker accessibility. This can complicate the database schema, as additional tables and metadata must be added to ensure sufficient performance.

The denormalization process is generally accepted by most users as it is viewed as a one-off task (performed once during the implementation of the system). However, this is rarely true since most database models can change throughout the lifetime of a system for various reasons, especially during the development phase. Once live, it can become increasingly difficult to update schema in a relational database system. The schema update process is known as migration and is the mechanism of structurally changing a relational database by applying a set of refactorings. This process can take weeks or months and is expensive, slow, and potentially risky. All these factors lead to a poor representation of the actual data and difficulty in maintaining the integrity during such changes.

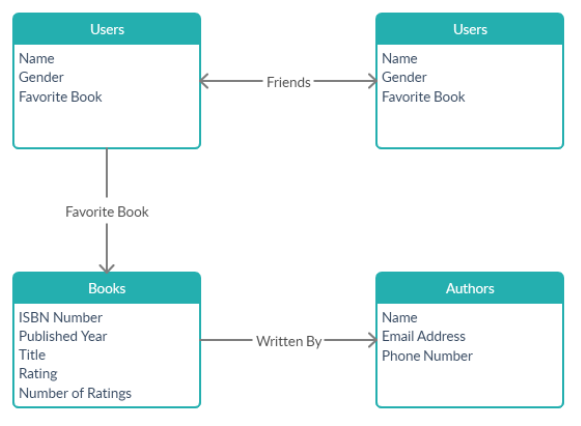
Most users will find graph databases to be a much more accurate and concise depiction of the system data. When implementing any database model, most users sketch the schema on paper. These original models almost always closely resemble the final graph models. That is because users sketch labels, attributes, and connections which directly correlate to the graph model implementation. Instead of transforming a domain model’s graph-like representation into tables, we enrich it, with the aim of producing an accurate representation of the parts of the domain relevant to our application goals [2]. This means the graph model does not need to go through a denormalization process like relational models.

When implementing graph models, the correct design decisions need to be made early in the process to ensure the support of rapid changes to the model which may have been initially overlooked. A couple techniques can be applied to ensure graph models provide this capability. The first is to read through the graph and ensure it makes sense. That is, to start at a random node and read each label and relationship as you work your way through the entire model. This will ensure the appropriate labels are attached to each node, and that relationships are documented in a way that traversing through the graph is simpler. The next technique is to describe the queries you expect the model to support. This will ensure the model supports the needs of end user’s whom will ultimately be using the model.

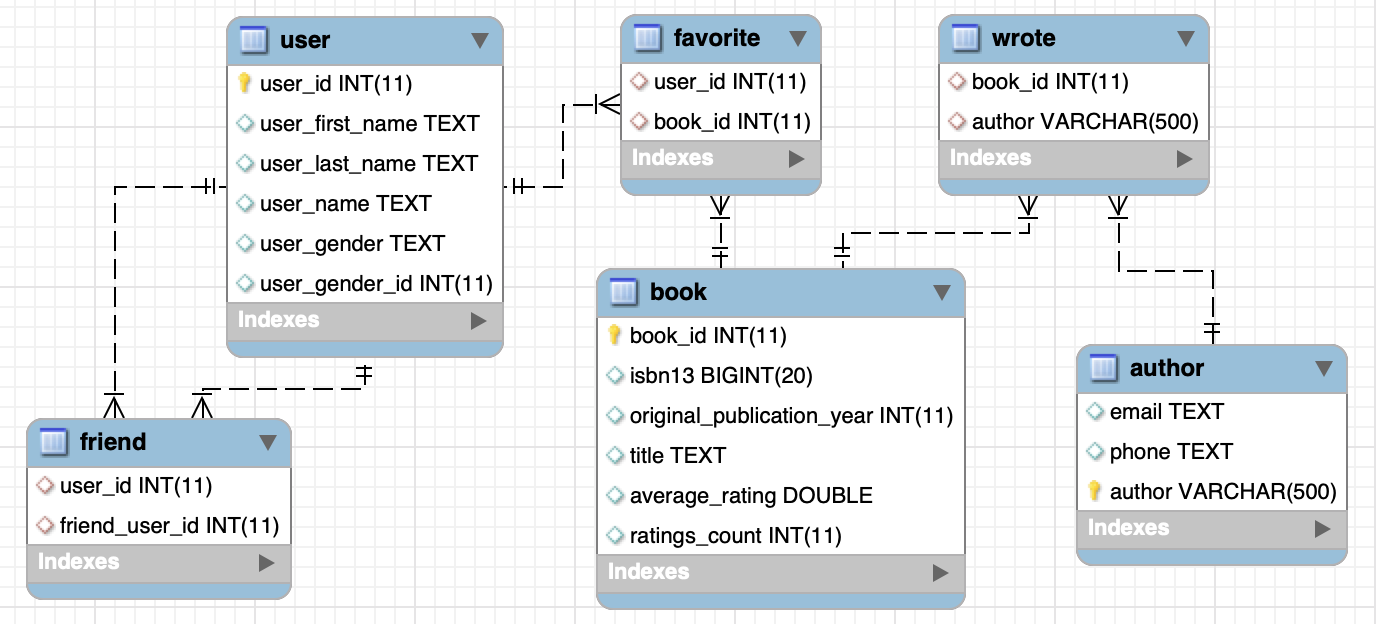
Cross domain support is another major advantage of graph databases which can allow for the discovery of new relationships within existing data. Since any node within the model can support relationships to many different nodes, some of which are in different domains, domains can be easily united. Cross domain relationships can also be easy to spot when reading through the graph, a technique that was previously described. Also, no metadata needs to be added to the graph model to support these relationships making their discovery organic. This can really enrich a system and provide additional resources that were not originally planned for or possibly overlooked.

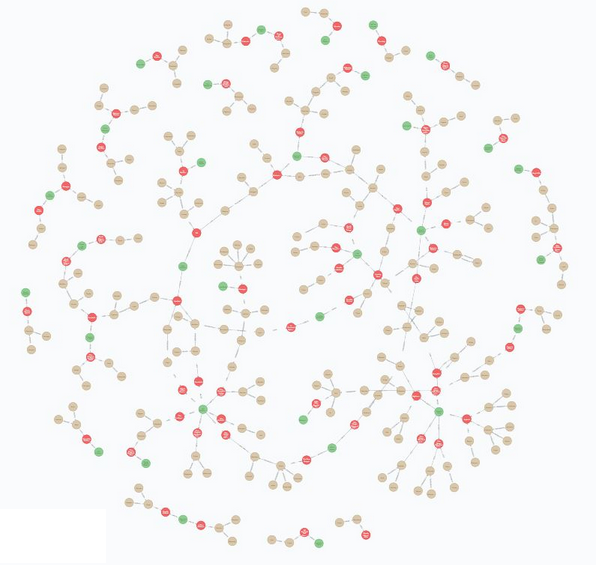
**VI. IMPLEMENTATION DETAILS**

For our experiment, we created a data model to reflect multiple depths. Our dataset is based off of user information from Goodreads, containing users, friends, their favorite books, and authors, shown in Figure 1 below. Our data set had 1000 users and a different favorite book for each user. There were also about 680 unique authors of the books. We added two friends for each user. From our previous research analysis, we hypothesized this would greatly show a performance difference between the two databases.

**Figure 1: Dataset Structure**

**Figure 2: MySQL Schema**



**Figure 3: Neo4j Data Model**

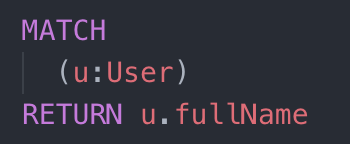
The expressiveness of Neo4j was noticeable when we implemented the ‘friends with’ relationship. In our graph database, we used a bidirectional relationship between the two users. If we did not want the relationship to be bidirectional, we could insert it as a directed relationship like the ones we used for “an author wrote a book” because “a book wrote an author” does not make any sense. However, in MySQL, we had two options of how to make the relationship bidirectional. We could either use a union in our queries because the user could either be entered in the user column or the friend column and we would have to check for both. The other option was to insert duplicate data into the friend table where we inserted it both ways. We chose to insert each friend pairing twice into the friend table since you would not be able to easily control the duplicates happening anyway.

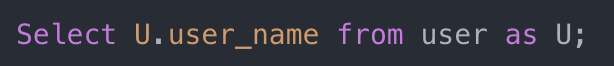
For Neo4j, we used the Neo4j Desktop Application and version 3.5.12 of the database. We utilized version 8.0.13 of MySQL. We used the Neo4j Browser and MySQL workbench to execute the queries and get the execution times. We attempted to query the Neo4j database from languages such as JavaScript, because the Neo4j browser is slow. The execution times returned by the Neo4j browser are not exact because they include deserialization and latency costs [10]. We attempted to query the Neo4j database from languages such as JavaScript. However, the Neo4j browser was is still faster than the times we got when using NodeJS. Also, the Neo4j browser is a helpful tool for visualization and management of the database. The next section details out our results.

**VII. EXPERIMENTAL RESULTS**

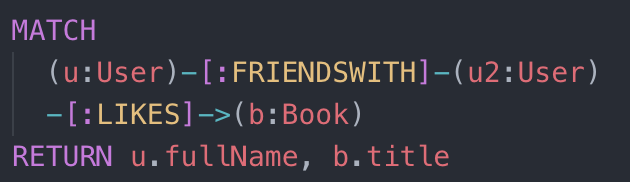
We ran two versions of each of the queries. One where we returned all of the attributes and another where only the most important attributes like the user names, the author names and/or book titles are returned. The following queries show the second version of the queries where only a few attributes are returned. The Cypher query language is able to define a relationship object and match or find nodes linked with the relationship. This made writing the query itself easy and concise, especially with queries that required many nodes and relationships. The MySQL queries, which were written in SQL, were much longer and required many joins.

***Query 1 (Users): Select all users***

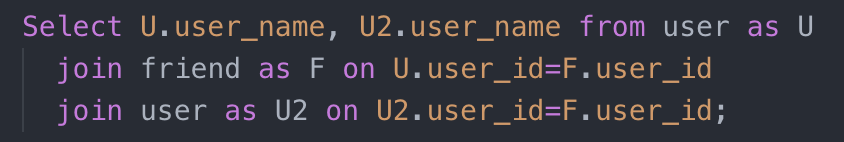
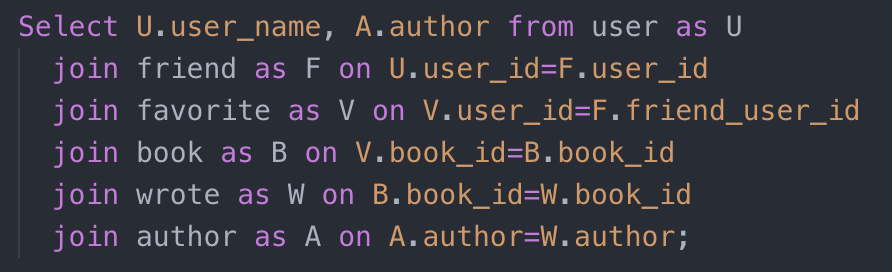
Cypher:

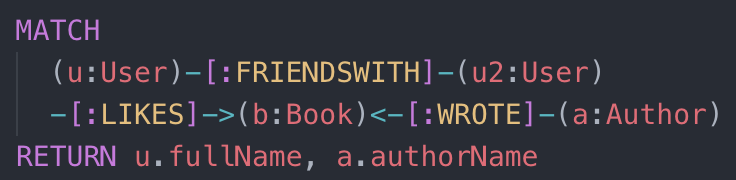
SQL:

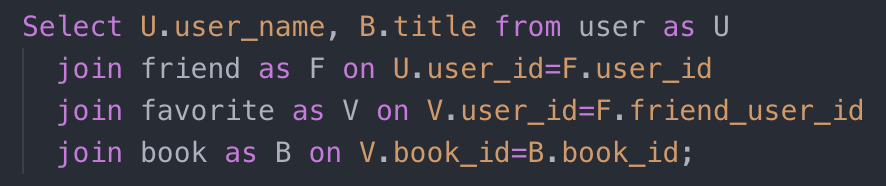
***Query 2 (Users and Friends): Select all users and their friends***

Cypher:

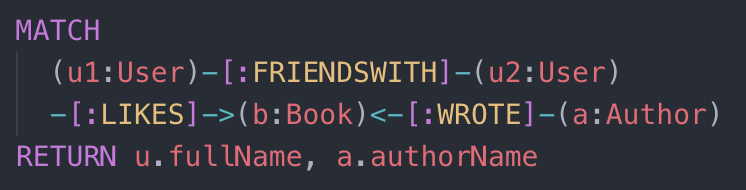
SQL:

***Query 3 (Book Suggestions): Select book suggestions for a user using their friends’ favorite books***

Cypher:

SQL:

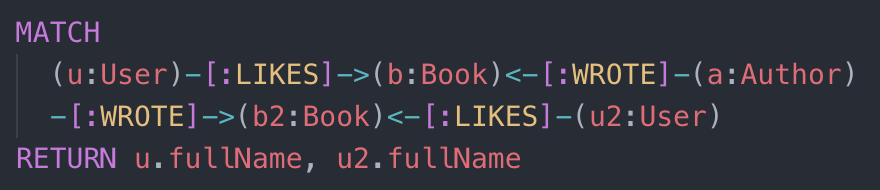
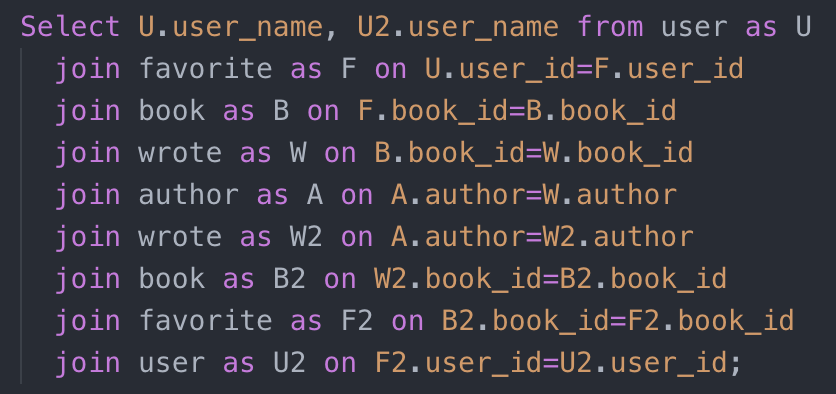
***Query 4 (Author Suggestions): Select author suggestions for a user using their friends’ favorite book’s author***

Cypher:

SQL:

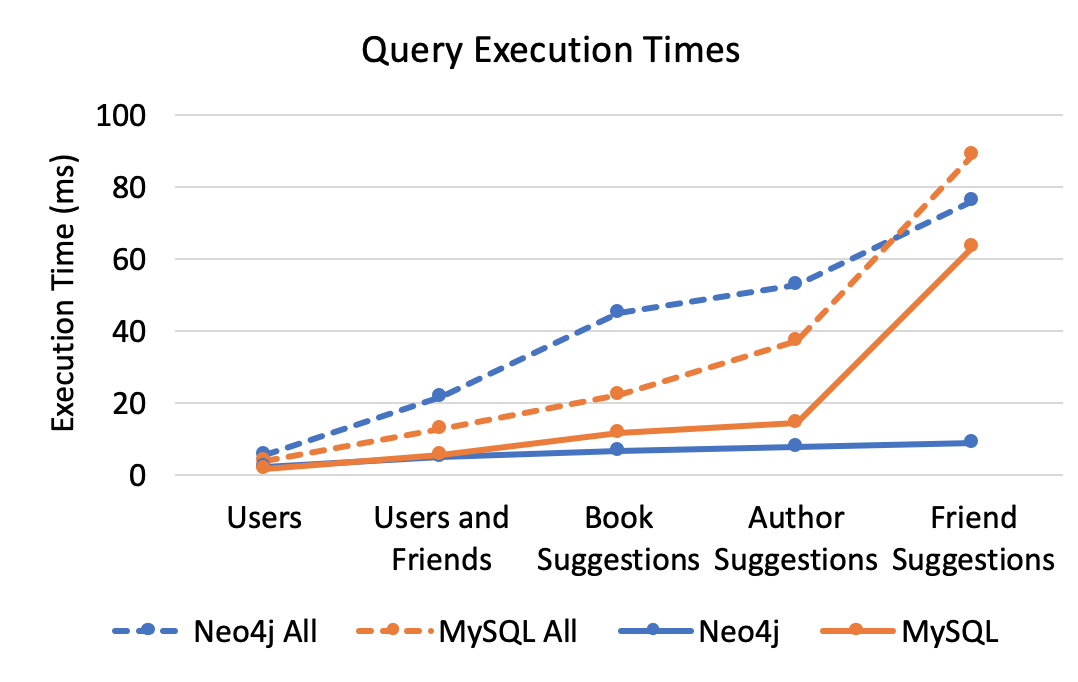
***Query 5 (Friend Suggestions): Select friend suggestions for a user using their favorite book’s author***

Cypher:

SQL:

**Table 1: Query Execution Times in Milliseconds**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Neo4j All Attributes** | **MySQL All Attributes** | **Neo4j** | **MySQL** |
| **Users** | 5.6 | 3.6 | 2.3 | 1.5 |
| **Users and Friends** | 21.5 | 12.7 | 5.0 | 5.6 |
| **Book Suggestions** | 45.0 | 21.9 | 6.5 | 11.4 |
| **Author Suggestions** | 52.5 | 37.3 | 7.5 | 14.2 |
| **Friend Suggestions** | 76.0 | 89.0 | 9.0 | 63.0 |

**Figure 4**

The ‘Neo4j All’ and ‘MySQL All’ queries, which are represented using the dotted lines, returned all the attributes from all of the entities and relationships involved in the queries. The solid lines represent the queries where only a couple relevant attributes are returned in the result.

For the queries that only return a couple relevant attributes, Neo4j times remain steady even in for final query. The additional cost of traversing farther through the relationships is small. MySQL was faster at returning all the users in the database. MySQL took an average of 1.5 milliseconds for the first query, returning all users, while Neo4j took about 2.3 milliseconds. However, the other four queries were significantly faster in Neo4j and the advantage to using a graph database was more apparent with each additional join required. The difference was especially noticeable with the last query, which required eight joins in SQL, where MySQL’s execution time quadruples when compared to the query before it which only required five joins, while Neo4j’s execution time only went up by a second and a half.

Returning all of the attributes was faster in MySQL for the first four queries, however in the final query that requires eight joins, the cost of the joins outweighed the time Neo4j takes to return all the attributes.

Next, we investigated the impact of having a higher number of relationships on both databases. We compared the select all users and their friends query on databases with different numbers of friend relationships. We used the original set up where each user has two friends and a new set up where each user is friends with all other users. We also created more versions of each database where each user has 20, 40 or 100 additional friends, plus the two original friends.

Each user has 2 friends (each user is friends with 0.2% of users):

~ 3.5K total friendships

Each user has 22 (20 additional + 2 original) friends: about 2.2% of users are connected

~ 22K friendships

Each user has 42 friends: about 4.2% of users are connected

~ 42K friendships

Each user has 102 original friends: about 12.2% of users are connected

~ 122K friendships

Each user is friends with every other user: 100% of users are connected

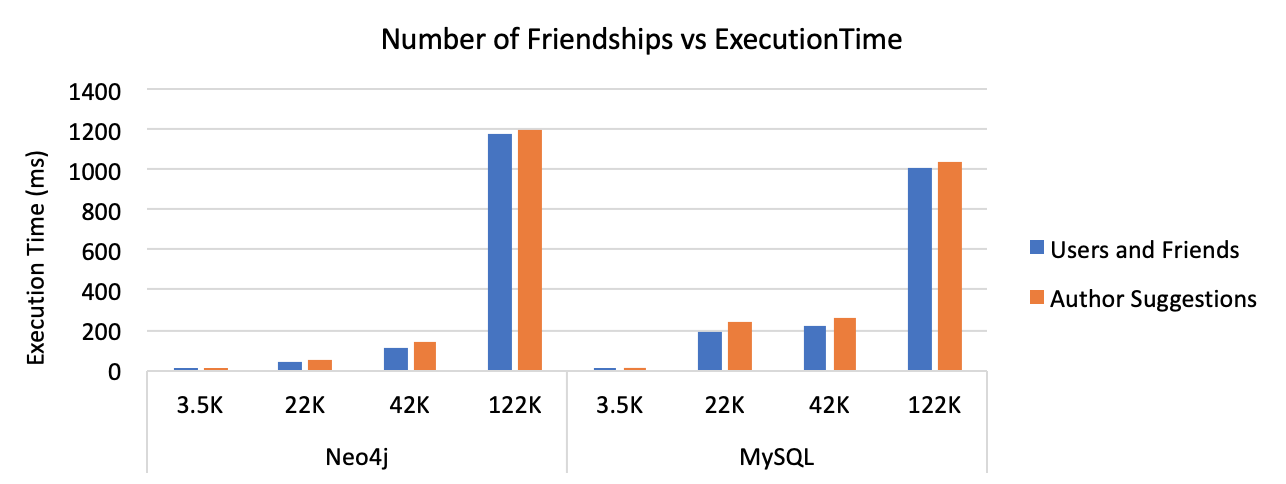
~ 998K friendships

*Query 1 (Users and Friends): Select all users and their friends*

*Query 2 (Author Suggestions): Select author suggestions for a user using their friends’ favorite book’s author*

**Table 2: Neo4j Execution Times per Number of Friendships in Milliseconds**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Neo4J** | | | | | **MySQL** | | | | |
| 3.5K | 22K | 42K | 122K | 998K | 3.5K | 22K | 42K | 122K | 998K |
| **Users and Friends** | 5.0 | 39.0 | 108.0 | 1,182.0 | 16,905.0 | 5.6 | 194.0 | 223.1 | 1,103.1 | 1,697,8.0 |
| **Author Suggestions** | 7.5 | 53.0 | 140.0 | 1,198.0 | 18,583.0 | 14.2 | 239.1 | 262.0 | 1,039.7 | 3,888.7 |

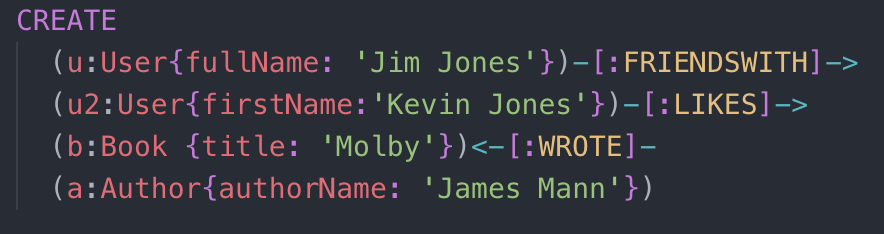
**Figure 5**

Neo4j executed the two queries faster when users had 2, 22, and 42 friends. MySQL was slightly faster when each user had 102 friends and significantly faster when all users were connected. This suggests that Neo4j is not optimized for such data intensive queries. Some possible reasons why Neo4j was not able to efficiently execute the queries when all of the users are connected are ineffective or not optimized storage and memory capabilities.

From experience with working on relational databases, we know that writing data with indices is expensive. Therefore, we devised to simple one record per table insert and compared that to inserting a single node and relationship for all the different types we have. The results are shown in Figure 6. We included all of the attributes in the following queries, but removed them in the images for clarity.

***Insert Query****: insert data across all nodes and relationships. Create a user, author, book and the relationships between them: friendship, written by and favorite book.*

Cypher:



SQL:

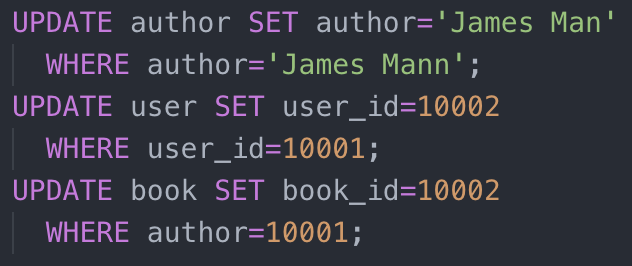


***Update Query****: update indexed attributes from all nodes: user, book, author. In Neo4j, we updated attributes that had indexes on them and in MySQL, we updated the attributes used for the primary and foreign keys. When updating the primary keys, we had the cascade option on to further optimize updating the indexes.*

Cypher:



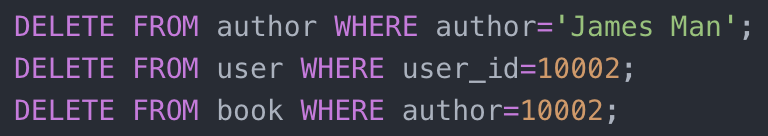
SQL:



***Delete Query****: delete all data that was inserted. This included deleting the user, author and book entities and the relationships between them. When deleting the primary keys, we also had the cascade option on to further optimize the deletions.*

Cypher:



SQL:

**Table 3: Query Times in Milliseconds**

|  |  |  |
| --- | --- | --- |
|  | **Neo4j** | **MySQL** |
| **Insert** | 2.0 | 6.4 |
| **Update** | 7.8 | 6.8 |
| **Delete** | 8.3 | 5.1 |

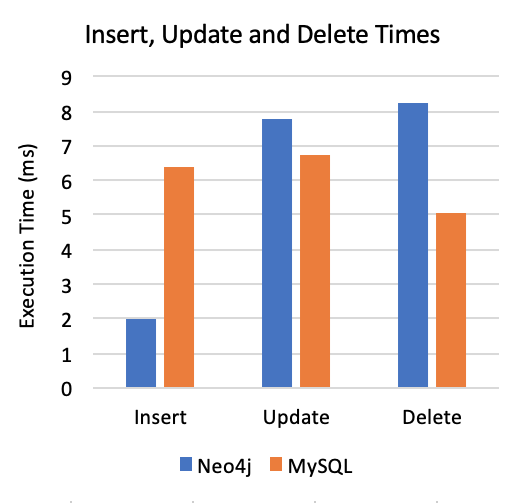
**Figure 6**

Figure 6 shows that the that insert query was faster in Neo4j and the update and delete queries were faster in MySQL. Neo4j executes creates the fastest because it does not require any searching. Finding nodes is a time-consuming operation in Neo4j, so updates and deletes are slower then creates. When deleting across all nodes and relationships we optimized the query using the detach delete command so it deleted the relationships by itself when we deleted the three nodes.

MySQL performed better than Neo4j when updating and deleting nodes. This is probably due to MySQL’s advanced, mature and highly optimized indices. The insert and update queries take about the same amount of time because they are doing similar things: scanning the indexes and checking for duplicates. The “on update cascade” and “on update cascade” options were used in MySQL and it helped reduce the update and delete queries by about half a millisecond. The delete took the least amount of time showing that writing is a costly operation in MySQL.

There are other considerations such as query optimization and the running environment that may have affected our results for Neo4j, because we were not familiar with Cypher or the best way to get execution times. We tried adding indexes on the graph database, but we did not see much improvement. We were more familiar with relational databases, so we were able to add more helpful indices and construct the schema more efficiently, and better optimize the queries.

**VIII. CONCLUSION**

From our experiment, we concluded that Neo4j is significantly faster at retrieving highly connected data and is easy to implement and query. There are no joins or indexes required. Graph databases are schema-free, which makes them highly flexible and expressive in the data it stores. However, there are still scenarios where a relational database is better than NoSQL options. They are more mature, secure and provide easier scaling. Relational databases are still better when the data has a relatively small and static number of relationships between tables. A Graph database in of itself is not a general replacement for a relational database, especially in companies and systems that have been around for a long time. The migration to a different technology stack, especially for databases, is always expensive. Both options have their advantages and use cases. However, with the type of dynamic, highly connected data being produced today by many applications and systems, graph databases are growing more popular. They are optimized for traversing relationships between data of any type.

**REFERENCES**

[1] David Reinsel, John Gantz, and John Rydning, "The Digitization of the World From Edge to Core", An IDC White Paper – #US44413318, Framingham, MA, Nov. 2018

[2] Ian Robinson, Jim Webber & Emil Eifrem, Graph Databases: New Opportunities for Connected Data. Sebastopol, CA: O’Reilly Media, Inc., 2015.

[3] “Neo4j Customers.” *Neo4j Graph Database Platform*, neo4j.com/customers/.

[4] "AWS Announces General Availability of Amazon Neptune", Business Wire, May 30, 2018. [Online]. Available: The Amazon Blog, https://press.aboutamazon.com/

[5] Robey Pointer, "Introducing FlockDB.", https://blog.twitter.com/engineering/en\_us/a/2010/introducing-flockdb.html

[6] Shalini Batra and Charu Tyagi, "Comparative Analysis of Relational And Graph Databases," International Journal of Soft Computing and Engineering (IJSCE), vol. 2, pp. 509-512, May 2012.

[7] Marta Patiño-Martínez and Diego Burgos-Sancho, " Snapshot Isolation for Neo4j," in Advances in Database Technology - EDBT 2016, 19th International Conference on Extending Database Technology, Bordeaux, France, March 15-16 2016, Open Proceedings, pp. 700-701.

[8] Chad Vicknair and Michael Macias, "A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective," ACMSE ’10, April 15-17, 2010, Oxford, MS, USA.

[9] Zahraa Mustafa Abdulrahman Al-Ani, “Narrow-band analyzer,” M.Sc., Department of Mathematics and Computer Science, Çankaya Univ., Ankara, Turkey, 2015.

[10] “Neo4j Browser User Interface Guide.” *Neo4j Graph Database Platform*, https://neo4j.com/developer/neo4j-browser/.