MY THOUGHTS ON POSTGRES vs. NEO4j

I experienced a handful of pros and cons in creating a database for SmallTown Hospital in Postgres and Neo4j. These pros and cons are discussed below.

The syntax required to create tables in Postgres is not overly complicated, which is a pro. As I did in Project 1, I utilized the ‘serial PRIMARY KEY’ feature to create unique identifiers in each table. This simple numerical incrementing process saved time and is easy to comprehend. Somewhat similarly, Neo4j does assign each node a unique identifier, but it is a six digit <id> that, to my knowledge, is only for Neo4j’s own usage under the hood. Because of its nature, I don’t think that it is vital that Neo4j have an autogenerating unique ID feature that the user can take advantage of, so I would not consider this a pro or con for Neo4j.

My experience with inserting the data into the tables I created in Postgres was positive. The Python script that I used to generate data outputted .txt files with insert statements, and Postgres read them in and populated the tables with no issue. Postgres’s performance during these inserts was reasonable, as I don’t think any of the inserts took more than 45 seconds. A great feature of Postgres that I took advantage of in this project is its ability to copy data in tables to CSV files with the COPY (SELECT \* FROM [table name] syntax. Using this feature, I was able to create seven CSV files that could be used to load data into Neo4j in a matter of seconds. With SmallTown Hospital’s data in CSV form, I assumed it would be straightforward to insert it into Neo4j. My assumption was incorrect. I had a very difficult time ensuring that my CSVs were stored in the designated ‘import’ folder, and that within the Cypher shell, I had Neo4j pointed at the right file path so that it could copy the CSV data into Neo4j. I then learned that each CSV had to be loaded in one at a time, and that in order to redirect the file path to each of the CSV files in my import folder, I had to exit the Cypher shell, change the file path, and then load the data. Because I ended up with eleven separate CSVs as part of my data load into Neo4j, this process was a pain and it was not intuitive to exit out of the shell after each import. From a performance standpoint, the data was loaded into Neo4j faster than the data was inserted into Postgres, so that was a plus. Also, despite the difficulty with pointing at the import folder, the fact that Neo4j allows for relationships to be created between columns in a CSV upon insert is fantastic. I took advantage of this feature, and it made the process of creating the necessary relationships between the nodes very easy. Once the nodes had been loaded, and the subsequent relationships created between those two nodes, the data could be queried. What relationships needed to be created took some thought, but to be able to get the data in and linked up this quickly was a huge pro for Neo4j.

Once the data and the accompanying schema/relationships were in place in both databases, it was time to retrieve patient information with queries. Because I utilized integers as unique identifiers for each doctor, patient, illness, and treatment, my queries in Postgres had to be slightly more involved to access the names of patients, doctors, illnesses etc. Even the queries that were just asking for a patient’s doctor required a join, which felt like overkill given the simplicity of the request. Along those same lines, in order for me to return a patient’s doctor, illness, and treatment by name in one query, I had to create three views, and then use those views as part of my query. I am not sure whether or not to consider this a pro or con for Postgres, as I was able to fetch the desired data, but it took a large amount of syntax to do so. On the other hand, the queries in Neo4j were much less verbose. I was able to return all information about a patient, including names, with a two-line query. Other than the convention of assigning nodes to a variable, the syntax for Cypher queries was pretty intuitive. It is also extremely powerful to be able to query for data based on node attribute, or relationships between nodes. The ‘other’ feature that can be utilized in conjunction with ‘<-->’ to find all of a patient’s illnesses or treatments was very useful as well. Another pro of the query experience in Neo4j was the availability of the web admin/data browser tool to display the results of the query in either graph or node format, and the ability to easily scroll through and rerun previous queries. From a performance standpoint, both databases returned the requested queries in less than a second, so one is not better than the other in this area.

Clearly, both Postgres and Neo4j have their advantages and drawbacks. One area, however, where I think both databases shine is in their documentation. Postgres’s documentation is thorough, not overly complicated, and easy to navigate. Neo4j’s is even more thorough, and it features tutorials on important topics like how to load data from a CSV and create nodes/relationships out of that data (which I took advantage of). I think that because Neo4j is non-conventional, and people may be weary of using it, that it goes above and beyond to make its documentation helpful.

Overall, I can see the benefits of using both databases. The relationships between the data of SmallTown Hospital was fairly homogeneous, so it was always going to be a good fit for a relational database like Postgres. Because of my experience with Postgres in project one, mapping the many to many relationships with intermediate tables and utilizing integers as unique identifiers made for a predictable, sound experience. I was apprehensive about using Neo4j to store and relate such homogenous data – it felt like its whiteboard nature wouldn’t fit well with SmallTown Hospital’s relatively predefined data structure. Its simplicity and freeform definitely took a while to wrap my head around, but once I established what relationships needed to be created, it was quick and easy to implement an effective database in Neo4j. In the end, my experience confirmed that Postgres is a reliable database that can manage this sort of data and opened my eyes to the unconventional yet powerful capabilities of a graph database like Neo4j.