CSE215: Design and Implementation of Database Systems
Course Project

Comparing Data Storage Strategies for Efficient Querying

Jackie Garcia
University of California, Santa Cruz
`{jgarc243@ucsc.edu}`

ABSTRACT

Graph databases have emerged recently as a response to the growing need for systems that can process and analyze large datasets. Relational databases have typically been used for this purpose, but their rigid structure and poor join performance can make it difficult to query large databases and analyze relationships between entities. By placing these relationships at the center of data storage design in graph databases, graph databases provide a powerful analytical alternative to relational systems.

The goal of this course project is to analyze the strengths, weaknesses, and tradeoffs between these two common data storage strategies - relational and graph - in terms of query execution time. I will implement the same datasets and run the same queries in different open-source databases in order to illustrate and analyze the differences in performance.

Ultimately, we can conclude that graph databases shine when handling table joins/relationships between different record types. Conversely, relational databases are more reliable - there is no large difference at importing new records based on the characteristics of that data. It is clear that graph databases are designed for relationship-centric datasets, but they can be modified to compete with relational databases in terms of query performance with access to optimizations such as indexes.

1. INTRODUCTION

Relational databases emerged in the 1970s as a response to the rigidity of the hierarchical and network database models that came before it[1]. It was revolutionary in that it allowed any set of records to be related to one another so long as they shared a common attribute - called a foreign key. This model eliminated redundancy and made data highly consistent - since most records exist in only one entry, it was no longer necessary to update every instance of a relationship[2]. Instead, entities that have many relationships can be updated in only one spot, since the relationships are not inherently stored in the database. These relationships only become visible when one queries the data on some join condition.

Recently, graph databases have been introduced as an alternative to the relational database model. Instead of requiring joins to connect records using primary keys, graph databases allow the user to create and store these relationships in the database itself [3]. This eliminates the need for joins entirely, since records (referred to as nodes) are connected to each other through graph edges, and could potentially lead to better query performance.

Whether this is true or not may depend on properties of both the dataset and the nature of the queries being performed. Some factors to consider include the size of the dataset, the size of intermediate query results, and the type of operations these queries perform. All of these must be taken into account when determining if a graph truly will perform better at querying than a relational database.

2. THE PROJECT

2.1 Goals

The goal of this project is to determine if a graph database can provide better query execution than a traditional relational database. We will achieve this by implementing the same datasets in both a relational and graph database. We will then execute the same queries in both systems, and note the differences in execution time. Additionally, we will use the EXPLAIN capabilities to understand why these systems perform differently, and where the strengths, weaknesses, and tradeoffs of either system come from.

2.2 The Systems

For the purposes of this project, I chose two similar, commonly used database systems. We will be comparing the querying capabilities of Postgres and Neo4j in execution time. Postgres is a relational database, while Neo4j uses a graph model. Both are open-source, and each comes with their own Data Query Language (DQL).

2.2.1 Postgres

Postgres is one of the leading, open-source, SQL relational databases. It was built by Michael Stonebraker in 1986 who, along with a team at UC Berkeley, added object-oriented capabilities to the existing relational database Ingres. In the mid-90s, Andrew Yu and Jolly Chen added SQL support to the system[4].

PostgreSQL ensures ACID (Atomicity, Consistency, Isolation, Durability) properties, has support for transactions, foreign keys, views, and indexing, among other querying tools[5].

Postgres does have support for more data types than Neo4j, but since I am testing and querying on the same attributes and columns, I do not see this being a confounding variable.

Postgres (Relational)	*Neo4j (Graph)*
Table	Node Type
Row	Node
Foreign Key	Relationship

Figure 1. Comparing Storage Types in Relational and Graph Data Models

2.2.2 Neo4j

Neo4j, or Network Exploration and Optimization 4 Java, is a graph database system. Version 1 was released in 2010 by Neo4j Inc, and the current version 4 was released at the start of 2020. Just like Postgres, Neo4j is ACID compliant, has support for transactions and indexing[6].

Instead of SQL, Neo4j has its own DQL: Cypher. Just like SQL, Cypher has the ability to select from different entry types (MATCH), filter results based on attribute values (WHERE), and return a query table (RETURN). Instead of storing records in tables however, Neo4j allows the user to create nodes of certain types. Cypher does not need to query on joins, since relationships can be queried instead.

4. THE DATASETS

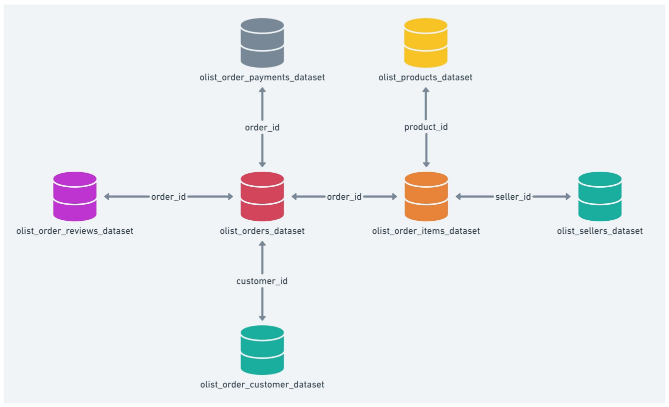


Figure 2. Brazilian E-Commerce Schema

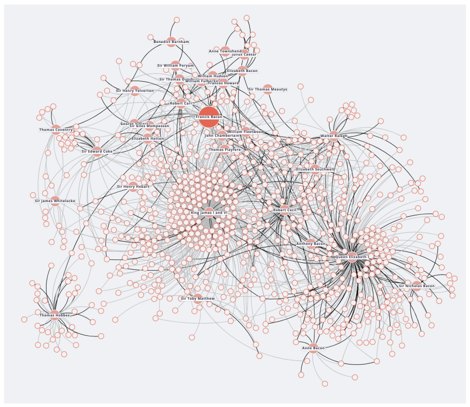


Figure 3. A Visualization of the Six Degrees Dataset

Both datasets used in this project were open-source and available on Kaggle.com. The first is a dataset that tracks orders from a Brazilian E-Commerce website[7], and the second is a social network entitled the Six Degrees Of Francis Bacon[8].

4.1 Brazilian E-Commerce

This dataset represents a Brazilian E-Commerce website. It has tables for reviews, orders, customers, sellers, products, and order items. Figure 1 above illustrates the schema, and the foreign keys that join each table. For example, the primary key customer_id from the Customer table is used to indicate which customer placed an order in the Orders table. The other foreign keys are order_id, product_id, and seller_id. For simplicity, the geolocation table from the original schema has been eliminated.

4.1.1 Relational Schema

For the relational schema, each CSV file from the original dataset became its own table. The data types for these tables were VARCHAR(50), INT, DECIMAL(6, 2), and TIMESTAMP. Each table was assigned its own primary key, and the join attributes were designated foreign keys as appropriate.

4.1.2 Graph Schema

For the graph schema, each table was represented as a node type. (Neo4j allows you to give the type of node a label for distinguishing types later on). By default, the data type of imported data in Neo4j is a string, so all VARCHAR(50) entries were left this way. All INT attributes were converted using toInteger(), and DECIMAL() attributes were cast toFloat(). Since timestamp support for the provided CSV format requires importing a library or transforming the data, these attributes were left as strings. This decision was made to not complicate calculating query execution time with the possible overhead of using a different conversion library.

The foreign keys were imported along with the other attributes of the table, and were then used to form the relationships between node types. This consisted of querying the two tables to form a cartesian product, and creating a relationship between nodes that had matching foreign key attributes. This is very similar to how joins are performed in SQL. In this case, however, these cartesian product joins only had to occur once to create the relationships, and all further queries could be done on the relationships without requiring a join.

Customers	99,441
OrderItems	112,650
OrderPayments	103,886
OrderReviews	99,985
Orders	99,441
Products	32,951
Sellers	3,095

Figure 4. Number of Records Per Table

Orders - OrderItems	112,650
Orders - OrderReviews	99,985
Orders - Customers	99,441
Orders - OrderPayments	99,985
OrderItems - Products	112,650
OrderItems - Sellers	112,650

Figure 5. Number of Relationships

4.2 Six Degrees of Francis Bacon

The Six Degrees of Francis Bacon dataset consists of two tables - people and relationships. There are 15,801 people with 171,408 relationships between them. Figure 2 illustrates the dense graph of relationships that connect the various people in the dataset.

4.2.1 Relational Schema

For the relational schema, I once again created a table for each type of record - People and Relationships. The data types were INT and VARCHAR(). Both tables had ID as their primary key. In addition, the relationships table had two foreign keys - person1ID and person2ID, that corresponded to the primary key ID of a person in the People table.

4.2.2 Graph Schema

For the graph schema, I created only one type of node for the People records. The data types mapped to those in the relational version - strings and integers. Instead of importing relationships as nodes, I read from the relationship CSV files line by line, fetched the people nodes with the corresponding IDs, and joined them with an edge. Neo4j allows you to store attributes inside the edge itself, so I stored the Relationship attributes inside the edge as opposed to in its own node.

```
LOAD CSV WITH HEADERS FROM ‘file’ AS row
MATCH (p1: People {ID:
   toInteger(row.person1ID)})
MATCH (p2: People {ID:
   toInteger(row.person2ID)})
CREATE (p1)-[:KNOWS
  {ID: toInteger(row.ID),
   startYearType: row.startYearType,
   startYear: toInteger(row.startYear),
   endYearType: row.endYearType,
   endYear: toInteger(row.endYear)
  }]->(p2);
```

Figure 6. CREATE Relationships code in Cypher.

```
CREATE TABLE Relationships (
      ID INT PRIMARY KEY,
      person1ID INT
         REFERENCES People(ID),
      person2ID INT
         REFERENCES People(ID),
      startYearType VARCHAR(5),
      startYear INT,
      endYearType VARCHAR(5),
      endYear INT);

COPY Relationships FROM 'file'
DELIMITER ',' CSV HEADER;
```

Figure 7. Same statement in SQL.

Already, you can start to see the main difference in performance between these two data storage models. Creating and importing data in PostgreSQL took roughly the same amount of time regardless of relationships between the records. This is an appealing property of the relational model, since it means that you can store large amounts of data with many relationships between records without a concern for creation execution time.

On the other hand, storing relationships in the graph database took roughly 30 times longer than it did in the relational database. This is because, in order to create a relationship, we had to scan the People table each time to find the correct node. This was a very expensive process, but fortunately, one that only needs to occur once to create all relationships.

5. MEASURING AND ANALYZING EXECUTION TIME

A series of queries were run in PostgreSQL and Neo4j. These queries were written in SQL and Cypher respectively. Postgres experiments were run on PostgreSQL version 11.11 on x86_64-apple-darwin16.7.0. Neo4j experiments were run on Neo4j Kernel version 4.2.1, enterprise edition. These databases were run locally on macOS version 11.2.1.

5.1 Creating Tables and Importing Data

5.1.1 Brazilian E-Commerce

In order to import data into PostgreSQL, it is first necessary to create the table schema using the system’s Data Definition Language (DDL), also known in SQL as the CREATE statement[9]. Because of this, Figure 8 illustrates the execution time of importing data by table in PostgreSQL in two ways - one that takes into account the time required to create the table, and the second only the time to import the data from the CSV file. In this case, the time necessary to create the table is so insignificant (roughly 50ms) that it is not considered costly overhead when comparing the two systems.

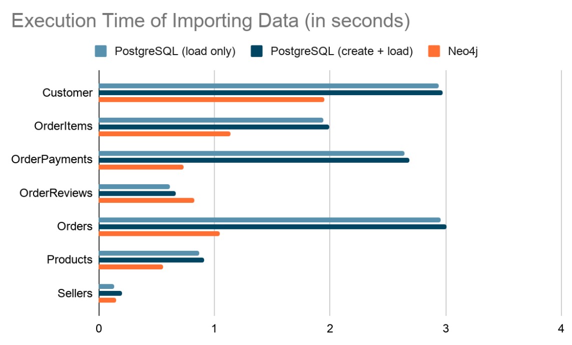


Figure 8. Execution Time of Importing Brazilian E-Commerce Data.

Neo4j instead allows the attributes of a node type to be set (CREATE) and loaded (LOAD CSV) at the same time[10], so only one execution time is considered. In general, Neo4j performs better than PostgreSQL at importing and storing new data. There are two exceptions: the OrderReviews table, where Neo4j is slower than both the CREATE TABLE and load operations in PostgreSQL, and the Sellers table, where Neo4j performs slightly worse than PostgreSQL's load, but still beats it out when we factor in the time to create the table as well.

Despite these exceptions, there was not a significant difference found between the time it took to import tables in either system.

5.1.2 Six Degrees of Francis Bacon

Unlike the E-Commerce example, there was a large difference in execution time between PostgreSQL and Neo4j when it came to importing the data for the Six Degrees dataset when it came to the Relationship table. This difference was so significant that it is almost impossible to see the PostgreSQL bars when put on the same chart (Figure 9). Figure 10 illustrates this same data, but adjusted to a log scale, in order to better demonstrate this difference. Figure 12 illustrates this difference in execution time as a percentage - which is where we can see that this added cost of fetching nodes results in data import that is 300 times slower than Postgres.

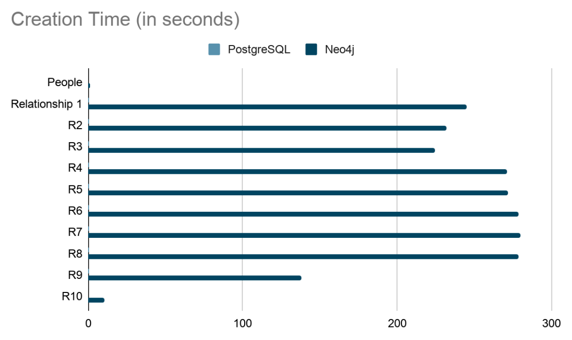


Figure 9. Comparing Creation Time for Six Degrees Data Set in Seconds

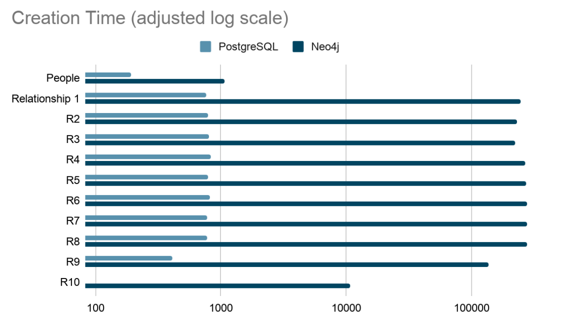
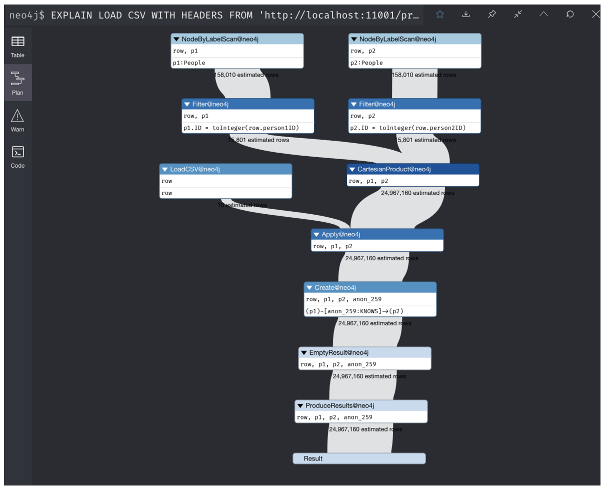


Figure 10. Comparing Creation Time for Six Degrees Data Set - Adjusted Log Scale

This experiment does represent the worst case scenario for Neo4j. By using the EXPLAIN keyword, Neo4j allows us to see the execution plan that the database chooses to perform this data import. This plan is shown in Figure 11. We can see that, in order to create relationships between people in the graph, it was necessary to scan the People table each time in order to find the node that corresponded to the correct ID for that relationship. This explains the significant cost, roughly 300 times more than the time it took to import the same data in PostgreSQL.

It is possible that by creating an index on the People ID, that the high cost of fetching nodes could be reduced. This is explored in a further section.



```
neo4j$ EXPLAIN LOAD CSV WITH HEADERS FROM 'http://localhost:11001/pr…
```

Figure 11. Execution Plan for Relationship Import

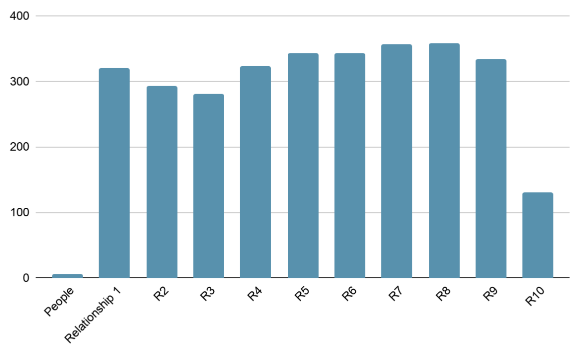


Figure 12. Speedup in PostgreSQL Import Against Neo4j

5.2 Calculating Degrees of Separation

5.2.1 Mapping A Single Person's Network

Execution Time (in seconds)	PostgreSQL	Neo4j
1	0.077	0.011
2	6.47	0.014
3	170.048	0.017
4	1321	0.11

Figure 13. Execution Time for Degrees of Separation Queries

One of the most common queries to run on the Francis Bacon network is, of course, the degrees of separation. This experiment consisted of choosing a person at random, and returning all people that are within X number of relationships to that individual. The first degree is simply all those that share an edge with this individual, the second degree could be described as the "friends-of-friends" graph, and so on. These experiments were run on randomly generated primary People IDs, and then averaged. Instead of calculating the full six degrees, this experiment set only goes up to 4 degrees of separation. This is because, without the inclusion of some index or other form of potential speedup, going past 4 degrees was very time consuming. In PostgreSQL, four degrees took almost half an hour to complete.

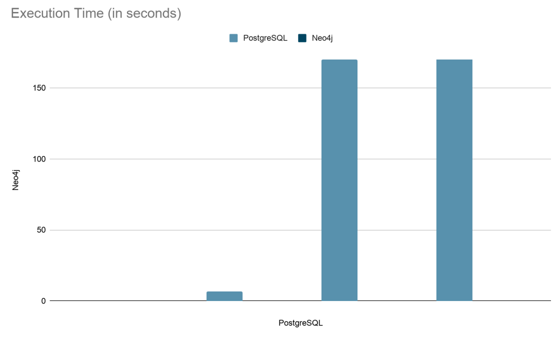


Figure 14. Execution Time for Degrees of Separation Queries in Seconds

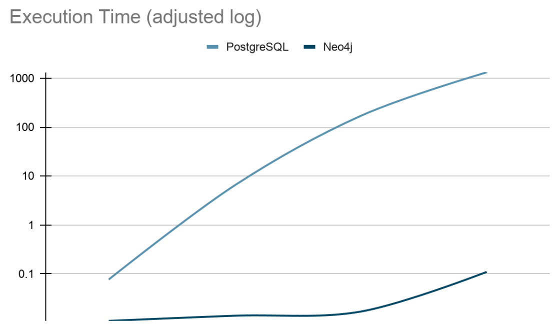
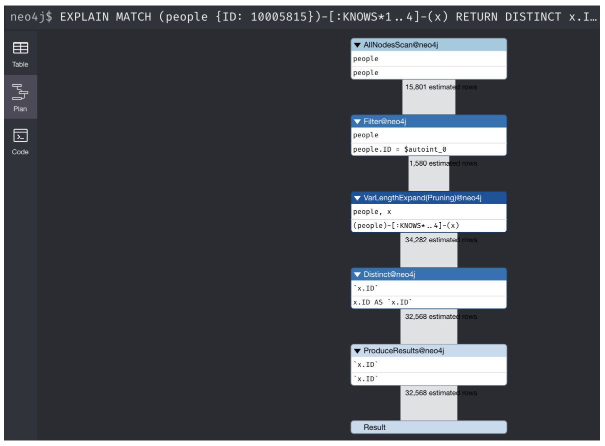


Figure 15. Execution Time for Degrees of Separation Queries Adjusted Log Scale

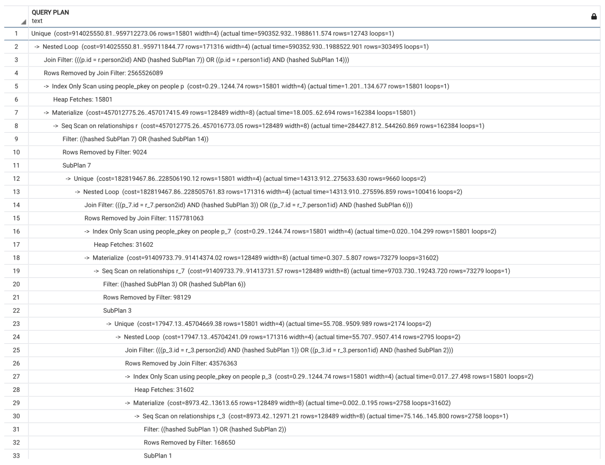
It is here where Neo4j shines. Since the relationship between nodes is already stored inside the database, querying for degrees of separation does not take much longer than a query for a regular attribute would. However, in PostgreSQL, this same query becomes very complex as the degrees grow. In order to find the first degree of separation, we must join the People and Relationship table and filter them. For subsequent degrees, this becomes more costly as we now must iterate through every person that was a result in the previous degree. Simply put, there is no cheap way to do this in a relational system. Much like in the previous set of experiments, the difference is so large, that it can only be visualized on the adjusted log graph.



```
neo4j$ EXPLAIN MATCH (people {ID: 10005815})-[:KNOWS*1..4]-(x) RETURN DISTINCT x.I…

AllNodesScan@neo4j
people
people
    15,801 estimated rows
Filter@neo4j
people
people.ID = $autoint_0
    1,580 estimated rows
VarLengthExpand(Pruning)@neo4j
people, x
(people)-[:KNOWS*..4]-(x)
    34,282 estimated rows
Distinct@neo4j
`x.ID`
x.ID AS `x.ID`
    32,568 estimated rows
ProduceResults@neo4j
`x.ID`
`x.ID`
    32,568 estimated rows
Result
```

Figure 16. EXPLAIN plan for 4 degrees in Neo4j



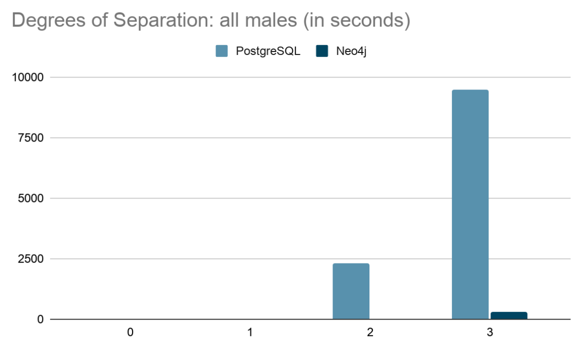
```
QUERY PLAN
text
Unique  (cost=914025550.81..959712273.06 rows=15801 width=4) (actual time=590352.932..1988611.574 rows=12743 loops=1)
  ->  Nested Loop  (cost=914025550.81..959711844.77 rows=171316 width=4) (actual time=590352.930..1988522.901 rows=303495 loops=1)
        Join Filter: (((p.id = r.person2id) AND (hashed SubPlan 7)) OR ((p.id = r.person1id) AND (hashed SubPlan 14)))
        Rows Removed by Join Filter: 2565526089
        ->  Index Only Scan using people_pkey on people p  (cost=0.29..1244.74 rows=15801 width=4) (actual time=1.201..134.677 rows=15801 loops=1)
              Heap Fetches: 15801
        ->  Materialize  (cost=457012775.26..457017415.49 rows=128489 width=8) (actual time=18.005..62.694 rows=162384 loops=15801)
              ->  Seq Scan on relationships r  (cost=457012775.26..457016773.05 rows=128489 width=8) (actual time=284427.812..544260.869 rows=162384 loops=1)
                    Filter: ((hashed SubPlan 7) OR (hashed SubPlan 14))
                    Rows Removed by Filter: 9024
                    SubPlan 7
                      ->  Unique  (cost=182819467.86..228506190.12 rows=15801 width=4) (actual time=14313.912..275633.630 rows=9660 loops=2)
                            ->  Nested Loop  (cost=182819467.86..228505761.83 rows=171316 width=4) (actual time=14313.910..275596.859 rows=100416 loops=2)
                                  Join Filter: (((p_7.id = r_7.person2id) AND (hashed SubPlan 3)) OR ((p_7.id = r_7.person1id) AND (hashed SubPlan 6)))
                                  Rows Removed by Join Filter: 1157781063
                                  ->  Index Only Scan using people_pkey on people p_7  (cost=0.29..1244.74 rows=15801 width=4) (actual time=0.020..104.299 rows=15801 loops=2)
                                        Heap Fetches: 31602
                                  ->  Materialize  (cost=91409733.79..91414374.02 rows=128489 width=8) (actual time=0.307..5.807 rows=73279 loops=31602)
                                        ->  Seq Scan on relationships r_7  (cost=91409733.79..91413731.57 rows=128489 width=8) (actual time=9703.730..19243.720 rows=73279 loops=1)
                                              Filter: ((hashed SubPlan 3) OR (hashed SubPlan 6))
                                              Rows Removed by Filter: 98129
                                              SubPlan 3
                                                ->  Unique  (cost=17947.13..45704669.38 rows=15801 width=4) (actual time=55.708..9509.989 rows=2174 loops=2)
                                                      ->  Nested Loop  (cost=17947.13..45704241.09 rows=171316 width=4) (actual time=55.707..9507.414 rows=2795 loops=2)
                                                            Join Filter: (((p_3.id = r_3.person2id) AND (hashed SubPlan 1)) OR ((p_3.id = r_3.person1id) AND (hashed SubPlan 2)))
                                                            Rows Removed by Join Filter: 43576363
                                                            ->  Index Only Scan using people_pkey on people p_3  (cost=0.29..1244.74 rows=15801 width=4) (actual time=0.017..27.498 rows=15801 loops=2)
                                                                  Heap Fetches: 31602
                                                            ->  Materialize  (cost=8973.42..13613.65 rows=128489 width=8) (actual time=0.002..0.195 rows=2758 loops=31602)
                                                                  ->  Seq Scan on relationships r_3  (cost=8973.42..12971.21 rows=128489 width=8) (actual time=75.146..145.800 rows=2758 loops=1)
                                                                        Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
                                                                        Rows Removed by Filter: 168650
                                                                        SubPlan 1
```

Figure 17. Part of the EXPLAIN plan for 4 degrees in PostgreSQL

This large difference in execution time can be better explained by looking at the query plans that both systems develop. Figure 16 shows the query plan for four degrees in Neo4j, and Figure 17 shows the first couple lines of the Postgres plan. From the Neo4j plan, we can see that all four degrees are calculated in the same step. This is not possible in Postgres, since relations between records are not stored in the database. Instead, the Postgres query plan requires calculating each degree recursively. Not only is this expensive, but it gets worse at every degree since the intermediate result sizes get larger each time.

5.2.2 Larger Networks

In order to test the capabilities of both systems on a large intermediate result size, I ran the same Degrees of Separation experiment, but on a set of records instead of just one person. This was done by searching for the network of nodes within three degrees of any node that had “male” as their gender.



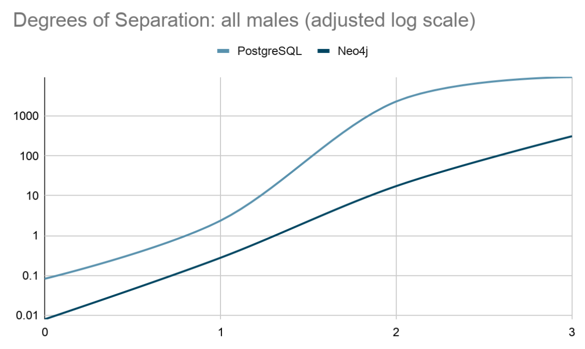


Figure 18. Degrees of Separation Execution Time in Seconds

Figure 19. Degrees of Separation Execution Time - Adjusted Log Scale

This furthered the difference in performance between Neo4j and PostgreSQL even more. At its worst (3 degrees), Neo4j was able to return a result in 312.119 seconds. Conversely, it took PostgreSQL 2 hours and 38 minutes to return the same result. Once again, this is mostly due to the fact that the relational model forced intermediate joins in order to find records that are related.

Figure 19 illustrates this execution time plotted on a log scale. We can see here that Neo4j scales uniformly as we calculate more degrees of relationships, but Postgres jumps drastically past the first degree. As we discussed in the previous section, this is because Postgres stores the intermediate query result tables, which get drastically larger at every degree.

5.3 Number of Table Joins

Based on the previous section, we can start to see how Neo4j performs better than PostgreSQL when it comes to queries that require joins. The degrees of separation queries require multiple levels of joins on the same table. This next section compares the performance of these systems at joining multiple tables.

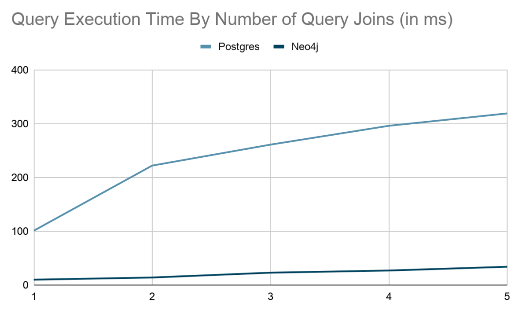


Figure 20. Execution Time Of Queries as Number of Table Joins Increases

This was done by creating a query that returned orderID from Orders - the table at the center of the Brazilian E-Commerce schema. To test new joins, each addition query added a join with a new table, and a single check on some attribute in that new table.

Once again, Neo4j handles the joins much faster than Neo4j can. From Figure 20, we see that Neo4j also scales better when we add more tables. The extra cost to add a new table and attribute check is very small compared to Postgres. Postgres still performs very well, but does have an increased query time every time a new table is added.

5.4 Indexing

As mentioned in section 5.1.2, Neo4j is not built to perform joins, and thus must fetch nodes one by one in order to create relationships. The average run time of importing a CSV file with 20,000 relationships is roughly 4 minutes without indexing. This high execution time came mostly from having to fetch People nodes based on their ID. Indexing may be able to speed up this process by adding a quicker access path to that primary ID. In order to test this theory, I created a new database, imported the People table again, and then create an index on the People attribute ID:

```
CREATE INDEX person_id_index FOR (n:People) ON (n.ID);
```

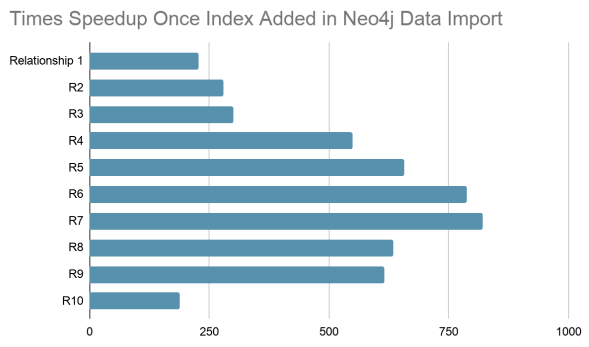


Figure 21. Import Speedup With Index

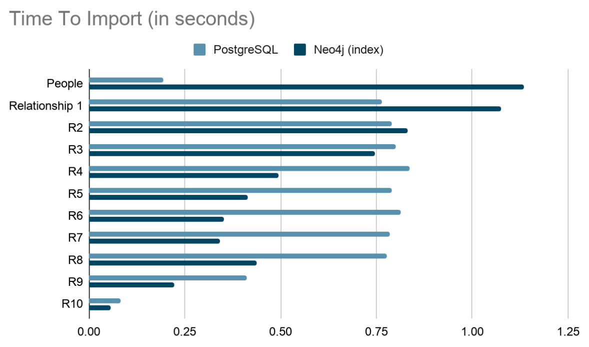
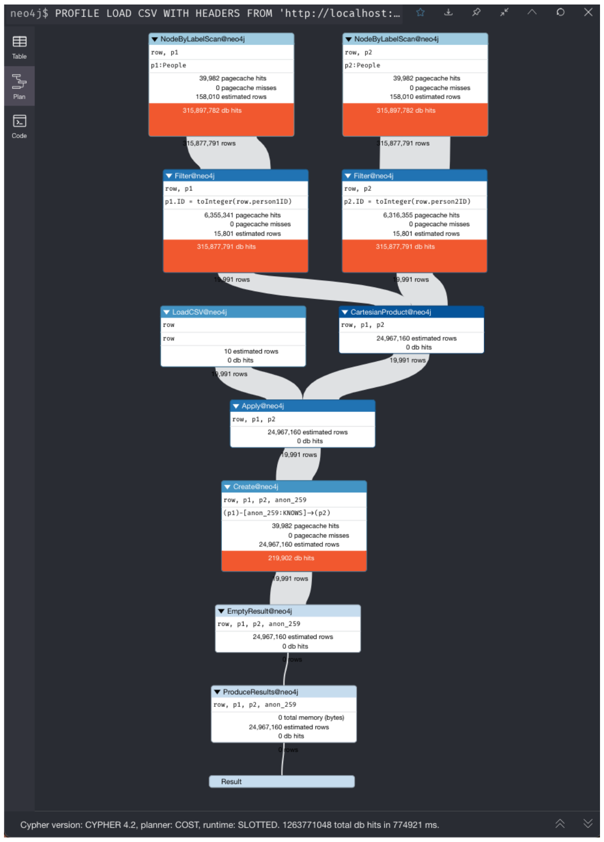


Figure 22. Time To Import - Postgres / Index

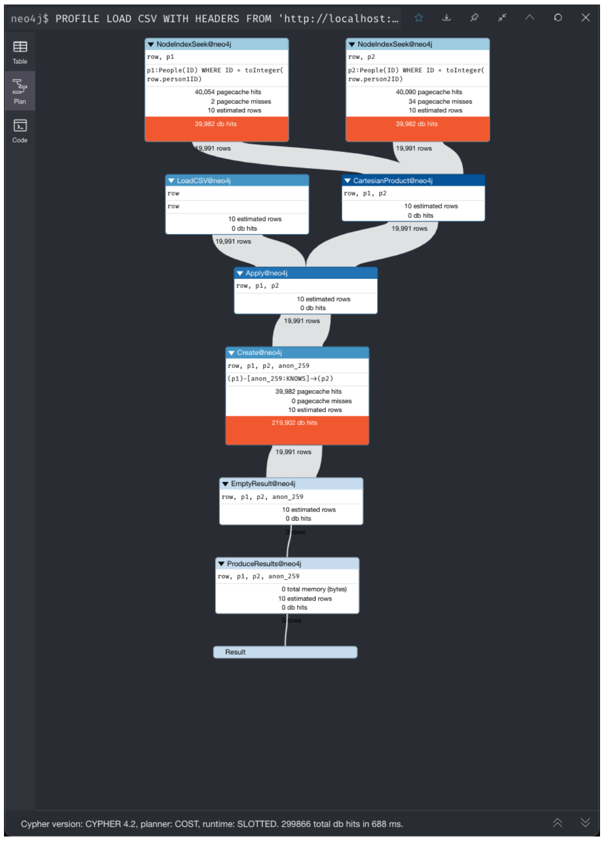
This index was created very quickly - taking only 192 milliseconds to complete. After this, I imported the relationships once again, using the same code, and the speedup was very apparent. Figure 21 shows this speedup - using the index resulted in an execution time between 220 and 820 times faster than the original import time. With this added index, Neo4j's performance now rivals PostgreSQL's, as shown in Figure 22. It is interesting to note that, unlike PostgreSQL, whose execution time remains constant as more relationships are created, Neo4j gets faster with each relationship.



```
neo4j$ PROFILE LOAD CSV WITH HEADERS FROM 'http://localhost:…
```

```
Cypher version: CYPHER 4.2, planner: COST, runtime: SLOTTED. 1263771048 total db hits in 774921 ms.
```

Figure 23. PROFILE for Relationship Import without index



```
neo4j$ PROFILE LOAD CSV WITH HEADERS FROM 'http://localhost:…
```

```
Cypher version: CYPHER 4.2, planner: COST, runtime: SLOTTED. 299866 total db hits in 688 ms.
```

Figure 24. PROFILE for Relationship Import with index

We can see why this is by profiling these queries. Neo4j allows us to analyze which operator is doing the most work/memory access in a query using the PROFILE keyword[11]. Figure 23 illustrates the computations performed to import the first 19,991 relationships without access to an index. Figure 24 shows the profile with access to the People ID index. Note that, because PROFILE does add overhead, the execution time with PROFILE is higher than the previous experiments I ran.

In the original plan, the People table and rows to be imported are joined in a cartesian product twice, for a total of 15,877,791 rows both times. This requires 315,897,786 database hits. Those results are then filtered to find the 19,991 nodes for Person 1, and the 19,991 nodes for Person 2 that will be joined by a relationship. This plan results in a total of 1,263,771,048 database hits, which explains the poor performance.

On the other hand, the index plan fetches the corresponding People nodes directly, which only requires 39,982 database hits, and creates this same filtered set of 19.991 People nodes on either side of the relationship. This plan only requires 299,866 database hits in total, since it fully eliminates the cartesian product between row and the People table that the non-index plan performs twice.

Ultimately, we can conclude that Neo4j was most likely not designed to handle cartesian products, and therefore does not optimize for these heavy joins. By allowing indexing, we can remedy the high cost of joining node types (or tables), with a high degree of success. With the index on join attribute, Neo4j is able to create relationships in roughly the same time that Postgres takes to import the relationships as its own table.

5.5 Writes

For this section, I tested the execution time of writes. I developed a series of queries that joined multiple tables, and observed execution time differences between running the read-only query, and running that same query but adding an update to the attribute returned.

```sql
SELECT r.reviewScore
FROM OrderReviews r
WHERE EXISTS (SELECT *
  FROM Orders o, Customer c
  WHERE o.orderID = r.orderID
  AND c.ID = o.customerID
  AND c.state = 'SP');

UPDATE OrderReviews
SET reviewScore = 4
WHERE EXISTS (SELECT *
   FROM Orders o, Customer c
  WHERE o.orderID =
OrderReviews.orderID
  AND c.ID = o.customerID
  AND c.state = 'SP');
```

Figure 25. A read and write SQL query.

```cypher
MATCH (Customer {state: 'SP'})
-[:ORDERS]-(o)
-[:ISREVIEWED]-(r)
RETURN r.reviewScore;

MATCH (Customer {state: 'SP'})
-[:ORDERS]-(o)
-[:ISREVIEWED]-(r)
SET r.reviewScore = 4
RETURN r.reviewScore;
```

Figure 26. A read and write Cypher query.

The Postgres run resulted exactly as one would expect. The read-only query averaged around 230 milliseconds, and the write query approximately 470 milliseconds. It is apparent that the cost of the added write roughly doubled the execution time.

The Neo4j run, however, produces some strange results. The equivalent read only query averaged 230 milliseconds, same as in PostgreSQL. However, when we add the write, the average runtime goes down to 15 milliseconds. Simply put, the write query performs better than the equivalent read query.

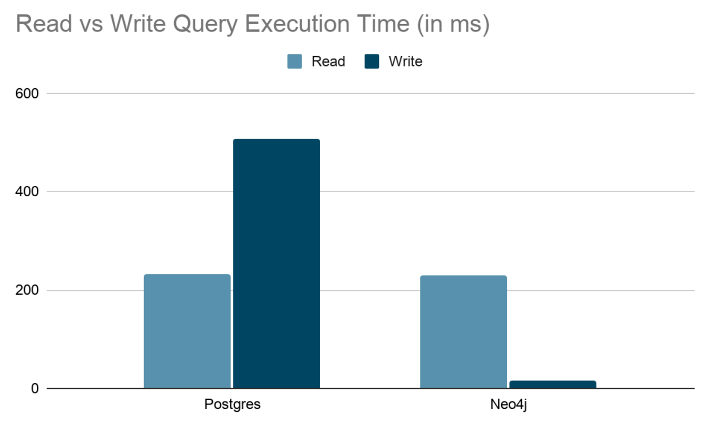
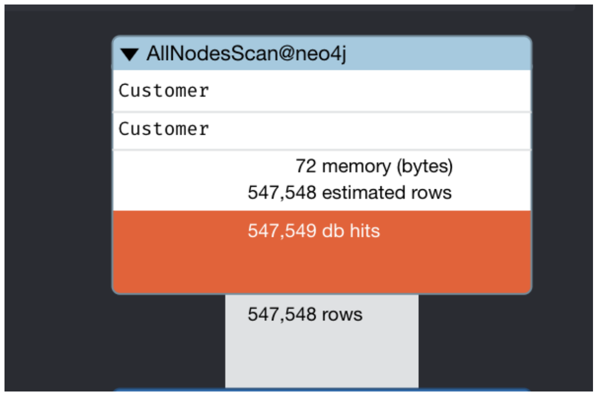
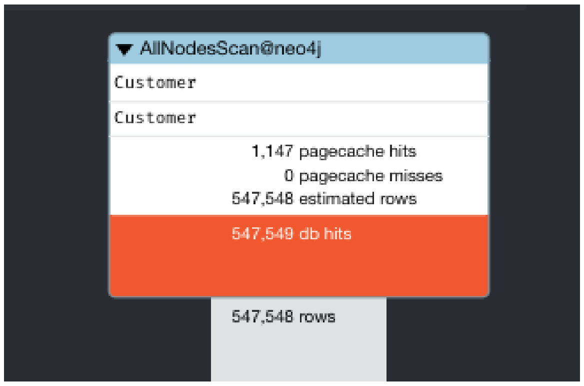


Figure 27. Execution time for read and write queries.

By comparing the profiles of the read and write query, it becomes clear why this is happening. The execution plans for the read and write queries are virtually identical, except for how they first access the Customer table. We can see in the figures below that in the read-only query, Neo4j reads nodes from reserved query memory. In the write query however, the system reads from a page cache. This is an interesting difference, and one that can be exploited for performance, since the size of the page cache can be manually tuned[12].

 ▼ AllNodesScan@neo4j Customer Customer 72 memory (bytes) 547,548 estimated rows 547,549 db hits 547,548 rows	 ▼ AllNodesScan@neo4j Customer Customer 1,147 pagecache hits 0 pagecache misses 547,548 estimated rows 547,549 db hits 547,548 rows
Figure 28. Step 1 of Read Query	***Figure 29. Step 1 of Write Query***

This choice in data access is why Neo4j is able to serve faster writes than reads, even with the same execution plan.

6. TAKEAWAYS

After performing these experiments, it is clear that Neo4j's excellent query performance comes from its ability to store relationships between records ahead of time, instead of having to calculate it at runtime. This explains why the graph system performs much faster than the relational database PostgreSQL when it comes to queries that involve joins - such as the Degrees of Separation experiments from Section 5.1 and the join queries from 5.3. Neo4j is also able to perform much faster writes (5.5), but that seems to be more specific to the implementation of page caching, instead of the fact that data is stored as a graph. There is some overhead involved at creation time (as seen in section 5.1) when it comes to these relationships, which can be very significant. This is likely because Neo4j is not designed to optimize joins, since it is expected that relationship edges will be queried instead. This can be remedied by adding indexes (5.4) that provide a better access path to the join attributes the relationships will be created on. Ultimately, we can conclude that Neo4j performed faster queries, while Postgres performed faster initial insertions into the database.

7. FURTHER WORK

In addition to the work in this paper, there are more areas that could be explored to analyze the relationship between data model design and system performance. Some ideas for future experimentation are listed below.

7.1 Further Update Testing

It is clear from the experiments performed that Neo4j generally performed better queries, and that the system's main weakness was in creating relationships between nodes. One of the most appealing aspects of the relational system is the fact that, when data changes, only one record must be updated, with no regard for the relationships that this record has. Thus, it would be helpful to see how Neo4j handles updates to its relationships. It is possible that this could be one area where it falls behind PostgreSQL's performance.

7.2 Compare Performance On Another Measure

This paper measures system performance in terms of query execution time. However, there are other factors one may consider when choosing a database system. Further exploration of these systems could focus on measuring performance in terms of CPU utilization, or space amplification, and see if the query execution performance of Neo4j comes at a cost of some other measure.

7.3 Testing Transaction Processing

Neo4j has support for READ COMMITTED transactions[13], which is a weaker isolation level that SERIALIZABLE, but was chosen in order to provide better transaction performance. PostgreSQL also supports READ COMMITTED, as well as other stricter isolation levels[14]. In addition to this, Neo4j allows the user to explicitly lock specific nodes and relationships using a Java driver. It would be interesting to continue to explore the performance comparison between Neo4j and PostgreSQL by testing both systems' transaction performance and runtime with concurrent transactions. Since Neo4j supports transactions using embedded Neo4j in a Java driver, extra care would be required to manage this overhead and produce comparable results between Neo4j and Postgres.

7.5 Introducing Another Storage System

While this paper only compares two storage models - relational and graph, there are many others that exist in database systems. It could be useful to introduce a third model, such as a key-value store, and implement the same datasets and queries to see how it performs in relation to the ones studied in this paper. This would further help to illustrate the effects that the data storage model of a database has on its query performance.

8. CONCLUSION

Graph databases provide a helpful alternative to the commonly used relational database system. They are designed with the relationships between records at the center of query processing. If one is trying to decide which database system to use, they may want to consider how often new data is inserted into the database, and how fast they would like to be able to query said data. If the data that they anticipate using will have large amounts of relationships between records, graph databases may be a good choice.

Alternatively, since graph databases are still relatively new, they do not always offer all the same capabilities as a classic relational database. Neo4j does not provide other transaction isolation levels besides READ COMMITTED, and has less data type support than Postgres. Overall, I think that the full power of graph databases is still yet to be explored. In the future, with a wider user base, and a support for more variety of needs, I believe they could someday replace the relational model as the most commonly used database system.

9. REFERENCES

[1] “Understanding the Network Database Model,” MariaDB. [Online]. Available: https://mariadb.com/kb/en/understanding-the-network-database-model/. [Accessed: 28-Feb-2021].

[2] IBM Cloud Education, “Relational Databases,” *IBM Cloud Learn Hub*, 06-Aug-2019. [Online]. Available: https://www.ibm.com/cloud/learn/relational-databases. [Accessed: 28-Feb-2021].

[3] “Why graph databases?,” 26-Aug-2019. [Online]. Available: https://neo4j.com/why-graph-databases/. [Accessed: 24-Feb-2021].

[4] K. Douglas and S. Douglas, “Introduction,” in *PostgreSQL: the comprehensive guide to building, programming, and administering PostgresSQL databases*, Indianapolis, IN: Sams Publishing, 2005.

[5] “What Is PostgreSQL?,” PostgreSQL Documentation, 11-Feb-2021. [Online]. Available: https://www.postgresql.org/docs/current/intro-whatis.html. [Accessed: 28-Feb-2021].

[6] “Neo4j Graph Data Platform,” *Neo4j Graph Database Platform*, 19-Jan-2021. [Online]. Available: https://neo4j.com/product/. [Accessed: 28-Feb-2021].

[7] Olist and André Sionek, “Brazilian E-Commerce Public Dataset by Olist.” Kaggle, 2018, doi: 10.34740/KAGGLE/DSV/195341.

[8] SDFB Team, Six Degrees of Francis Bacon: Reassembling the Early Modern Social Network. www.sixdegreesoffrancisbacon.com (August 29, 2017).

[9] PostgreSQLTutorial.com, “Import CSV File Into PostgreSQL Table,” *PostgreSQL Tutorial*. [Online]. Available: https://www.postgresqltutorial.com/import-csv-file-into-posgresql-table/. [Accessed: 24-Feb-2021].

[10] “Tutorial: Import Relational Data Into Neo4j - Developer Guides,” Neo4j Graph Database Platform. [Online]. Available: https://neo4j.com/developer/guide-importing-data-and-etl/. [Accessed: 24-Feb-2021].

[11] “Query tuning,” Neo4j Cypher Manual. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/query-tuning/query-options/#how-do-i-profile-a-query. [Accessed: 28-Feb-2021].

[12] “Neo4j Performance Tuning,” Neo4j Developer Guides. [Online]. Available: https://neo4j.com/developer/guide-performance-tuning/#memory-config. [Accessed: 28-Feb-2021].

[13] J. Rocha, “Neo4j current transaction commit process order,” *Neo4j Knowledge Base*. [Online]. Available:

[14] “Transaction Isolation,” *PostgreSQL Documentation*, 11-Feb-2021. [Online]. Available: https://www.postgresql.org/docs/9.5/transaction-iso.html. [Accessed: 28-Feb-2021]. https://neo4j.com/developer/kb/neo4j-current-transaction-commit-process-order/. [Accessed: 28-Feb-2021].

10. CODE

All code used to create and query tables for these experiments can be found at: https://github.com/jgarc243/CSE215