

Comparing Neo4j with PostgreSQL

SCS 3252:017 **Big Data Management Systems & Tools**

Kevin Carmona-Murphy, December 3rd 2019

Introduction

Choosing a database to pair with one's backend application is an important decision every developer must face when architecting the solution for their problem domain. Nowadays, there are a variety of different database products that offer advantages to apps using certain workflows or which have certain requirements. The traditional relational database paradigm has been the mainstay of the database world since the late 1970's, but newer technologies including column-oriented storage and NoSQL DBs have appeared on the scene in order to solve limitations in dealing with data that is not strictly accessed as rows in a table.

Graph databases have come into popularity beginning in the early 2000's. Graph databases emphasize the relationships between entities in the DB and not just the data itself. Queries on data that are highly interconnected benefit from the index-free adjacency properties (explained later) that some native graph database engines provide. Popular problem domains commonly modelled with graph databases include social networks, route planning and search algorithms.



Figure 1: Source: <https://www.bmc.com/blogs/neo4j-graph-database/>

In this report, I'll be comparing the performance of specific queries performed in Neo4j, currently the world's most popular graph database offering, with PostgreSQL, a stable, high-performing RDBS. I will be using [2015 Flight Delays and Cancellations](#) dataset provided by the US Department of Transportation on Kaggle as sample input for the queries. I'll demonstrate that it is advantageous to use Neo4j for queries that can easily be modelled to the problem domain, while PostgreSQL is the better choice for simple row-by-row based queries.

Objectives

The objectives of this report are threefold:

1. Briefly explain the difference between a native graph database and a relational database engine
2. Describe the sample data and how it is possible to model it in such a way that graph queries can be made in a performant manner
3. Examine why the queries yield the outcomes they give based on how each database engine fetches the data

Findings

Graph Database vs RDBMS

What's with all the hype about graph databases these days? As will be evidenced in the "Timing the Queries" section of this report, a GDBMS (Graph Database Management System) can outperform relational databases when the relationships between data are of more importance than the data itself, and when the dataset has been modelled in such a way that queries can take advantage of these relationships.

Graph databases can loosely be grouped into two categories: **native and non-native**. Non-native graph databases rely on a persistence layer supported by a columnar, NoSQL, or typical relational database technology. Non-native graph databases usually expose a query language and methods that make it easier to query data that can be modelled as graphs. However, these databases must use algorithms to transform row-by-row data that may be stored in disparate locations into the output format stipulated by the query language. This can result in inefficiencies and scalability problems when compared to native graph databases. An example of a popular non-native graph engine is OrientDB.

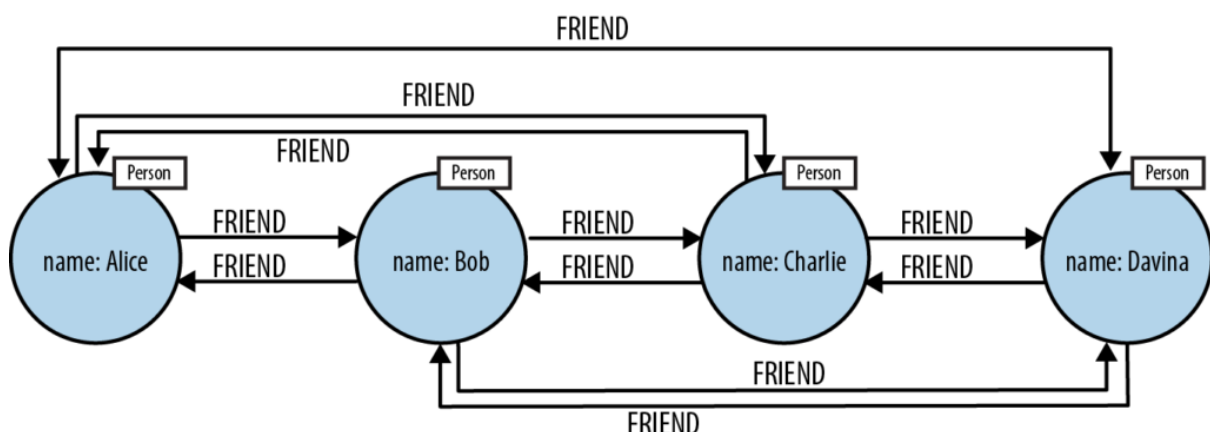


Figure 2: Illustrating the directed relationships present in a graph database. Source: <https://dzone.com/articles/graph-databases-for-beginners-native-vs-non-native>

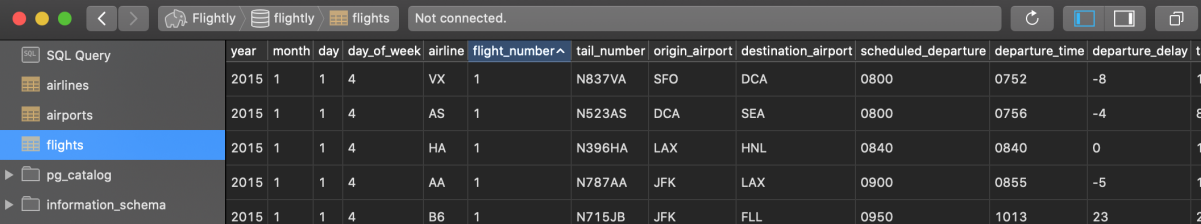
Native graph databases on the other hand exhibit **index-free adjacency**. This means that the relationships between individual nodes in a dataset are stored as pointers to other

nodes. Traversing from one node to another is done in $O(1)$ time as the DB engine simply follows the pointer to the next node, where its properties are then immediately available. A non-native GDB or a typical row-store RDB finds nodes in $O(\log(n))$ time because it must use indexes. An index lookup typically uses an algorithm like binary search, hence its $O(\log(n))$ performance. The term index-free adjacency stems from the fact that a native graph database eschews the use of indexes.

Neo4j is an example of a native graph database because none of its internal components are substituted with non-native technologies, meaning that it benefits from lightning fast lookups. It can scale to millions, even billions of nodes because lookup times depend only on the number of nodes searched in the query, and not the total number of nodes present in the database. In a relational database on the other hand, as the amount of data increases, the time it takes to search the index also increases.

Modelling the Sample Data

The sample data provided by the US Department of Transportation was chosen because it can be nicely modelled in such a way that makes it advantageous to store in a graph database and therefore produce favourable results when compared against row-store DBs. The properties that make this true are the following: the data provided is split into multiple tables, the data domain can be modelled as a directed graph, and the data is huge.



	year	month	day	day_of_week	airline	flight_number	tail_number	origin_airport	destination_airport	scheduled_departure	departure_time	departure_delay	tail_number
airlines	2015	1	1	4	VX	1	N837VA	SFO	DCA	0800	0752	-8	1
airports	2015	1	1	4	AS	1	N523AS	DCA	SEA	0800	0756	-4	8
flights	2015	1	1	4	HA	1	N396HA	LAX	HNL	0840	0840	0	1
pg_catalog	2015	1	1	4	AA	1	N787AA	JFK	LAX	0900	0855	-5	1
information_schema	2015	1	1	4	B6	1	N715JB	JFK	FLL	0950	1013	23	2

Figure 3: Quick sample of the flights table from sample dataset

The data is split into multiple tables: The 2015 Flight Delays and Cancellation dataset is split into 3 tables: *airlines*, *airports*, and *flights*. The smallest table, *airlines*, simply relates the IATA (International Air Transportation Association) airline codes with their full names. The *airports* table also does this, but adds extra fields such as *latitude*, *longitude* and *state*. Finally, the *flights* table, which is by far the largest, includes the columns shown in the above figure, along with many others such as actual arrival times, and whether the flight was cancelled or diverted.

The reason why it is important to acknowledge the split up of the sample data into multiple tables is that a relational database incurs a performance penalty at query time when performing a *JOIN* operation. Graph databases on the other hand can be pre-optimized using *MATCH* queries (shown in Appendix 1) to link together data represented in separate tables. This pre-optimization will be shown to yield speedier queries.

The data domain can be modelled as a directed graph: The sample data can be modelled as a series of entities representing the different stakeholders involved in operating a flight.

These entities include the source and destination **airports**, the **flight** itself, the **tail** (unique identifier of the aircraft being used for the flight), and finally the **airline** operating the flight. Between these entities we can add directed relationships that give meaning to how they are connected. The English statement “Flight 459540 has a departure from ORD, flies to LAX, is operated by United Airlines using the aircraft registered as N33209” can be transformed very easily into the below representation in Neo4j:

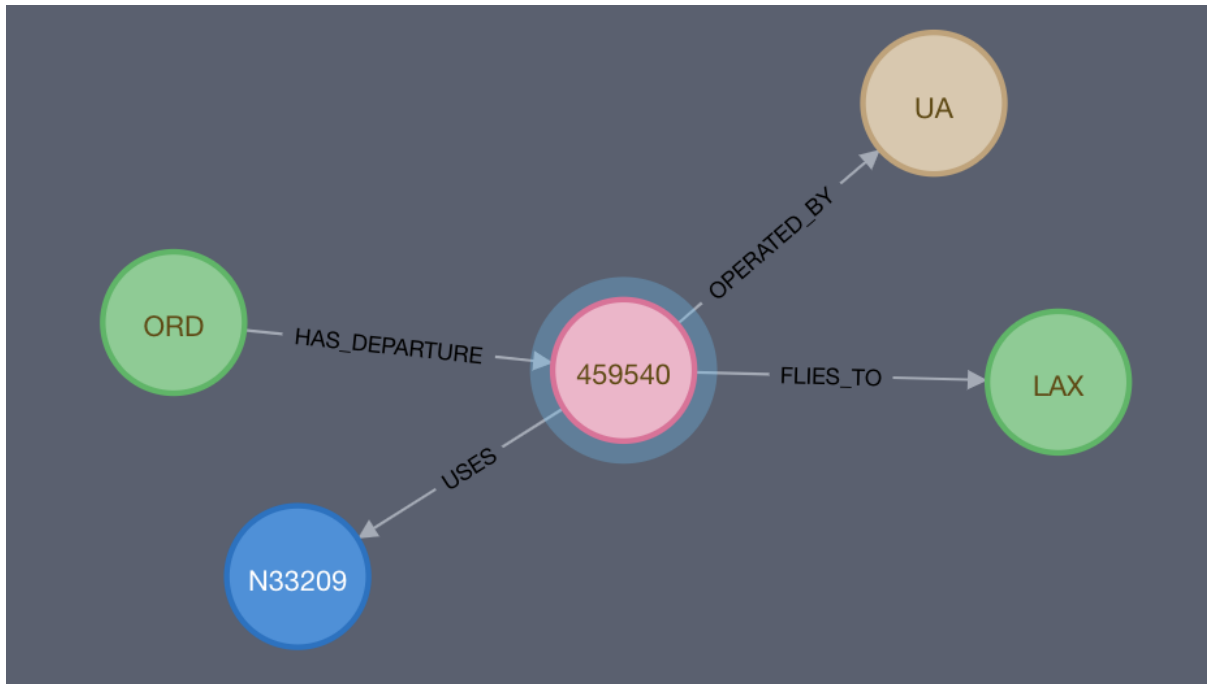


Figure 4: Actual visual representation of a Cypher query run in the Neo4j Desktop Browser

The data is huge: Neo4j is a technology suited for dealing with massive amounts of data. Although it doesn't support sharding or operating in a clustered environment like Hadoop or Spark, because of its index-free adjacency, it can scale really well. *As a side note about the size of the data, the *flights* table alone is over 500MB in size and on a standard 2013 Macbook would take over 20 minutes just to load in all of the data! I have therefore imported only the first 20000 rows into both the Neo4j and PostgreSQL databases for the purposes of this project. That equates to approximately 2 days worth of flight data.

Timing the Queries

This section presents seven queries that operate on the sample dataset, written in both the Cypher and Postgres SQL query languages. Each query was run 200 times consecutively, a total of 3 distinct times, with their times measured and averaged for each batch of runs. The outputs of both queries were matched against each other to prove that the queries in each language return the same result.

Please see the [following notebook](#) for the complete code used to generate the timings.

Single table queries

The following two queries show that for single table lookups, PostgreSQL performs better than Neo4j. This is expected, since an RDBMS's bread and butter is the quick execution of SELECT queries. Indeed, the first query runs in about half the time in Postgres, while the second query is even faster, taking advantage of Postgres' robust handling of on the fly type conversion.

1) All flights out of Chicago O'Hare

Cypher:

```
MATCH (f1:Flight {origin_airport: 'ORD'})
RETURN f1.flight_number
```

Postgres SQL:

```
SELECT flight_number from flights where origin_airport = 'ORD'
```

Attempt	1	2	3	Avg
Neo4j	3.949004316s	3.65151617s	3.902102932s	<u>3.78023859s</u>
PostgreSQL	1.544277276s	1.52129229s	2.215063601s	<u>1.98720932s</u>

2) All flights with a delay of more than 100 minutes

Cypher:

```
MATCH (f1:Flight)
WHERE toInteger(f1.departure_delay) > 100
RETURN DISTINCT f1.flight_number
```

Postgres SQL:

```
SELECT flight_number from flights WHERE CAST (departure_delay AS
INTEGER) > 100
```

Attempt	1	2	3	Avg
Neo4j	6.849503843s	6.3155989s	6.348862873s	<u>6.51349234s</u>
PostgreSQL	2.481023282s	2.2212819s	2.230398828s	<u>2.29819238s</u>

Using Relationships

The next 3 queries illustrate that Neo4j outperforms Postgres when the MATCH statement can benefit from the use of the node relationships that were mentioned earlier in the report (see Appendix 1). Now, these queries are tapping into the full potential of index-free adjacency. Once Neo4j anchors onto the *atl* and *clt* Airport nodes, the task of finding the *fl* node because as trivial as following all outgoing *HAS_DEPARTURE* and *FLIES_TO* relationships and seeing where they meet.

3) Flights departing from Atlanta and arriving in Charlotte

Cypher:

```
MATCH (atl:Airport {iata: 'ATL'})-[:HAS_DEPARTURE]->
      (fl:Flight)-[:FLIES_TO]->(clt:Airport {iata: 'CLT'})
RETURN fl.flight_number
```

Postgres SQL:

```
SELECT flight_number from flights WHERE origin_airport = 'ATL'
AND destination_airport = 'CLT'
```

Attempt	1	2	3	Avg
Neo4j	0.564938849s	0.48479266s	0.64348094s	<u>0.57234933s</u>
PostgreSQL	1.665589719s	1.80806948s	1.72529084s	<u>1.73929839s</u>

4) Flights departing from Chicago and arriving in LAX operated by United Airlines

Cypher:

```
MATCH (ord:Airport {iata: 'ORD'})-[:HAS_DEPARTURE]->
      (fl:Flight)-[:FLIES_TO]->(lax:Airport {iata: 'LAX'}),
      (fl)-[:OPERATED_BY]->(ua:Airline {iata: 'UA'})
RETURN fl.flight_number
```

Postgres SQL:

```
SELECT flight_number from flights WHERE origin_airport = 'ORD'
AND destination_airport = 'LAX' AND airline = 'UA'
```

Attempt	1	2	3	Avg
---------	---	---	---	-----

<i>Neo4j</i>	1.40462679s	1.0755725s	1.62699902s	<u>1.37639239s</u>
<i>PostgreSQL</i>	2.08356323s	1.5564846s	1.89876629s	<u>1.78923831s</u>

- 5) Flights departing from Chicago and arriving in LAX operated by United Airlines with Tail Number N33209

Cypher:

```
MATCH (ord:Airport {iata: 'ORD'})-[:HAS_DEPARTURE]->
      (f1:Flight)-[:FLIES_TO]->(lax:Airport {iata: 'LAX'}),
      (f1)-[:OPERATED_BY]->(ua:Airline {iata: 'UA'}),
      (f1)-[:USES]->(tl:Tail {number: 'N33209'})
RETURN DISTINCT f1.flight_number
```

Postgres SQL:

```
SELECT flight_number from flights WHERE origin_airport = 'ORD'
AND destination_airport = 'LAX'
AND airline = 'UA' AND tail_number = 'N33209'
```

Attempt	1	2	3	Avg
<i>Neo4j</i>	1.27102009s	1.03612895s	1.0988422s	<u>1.17559017s</u>
<i>PostgreSQL</i>	1.66821703s	1.51656632s	1.5901762s	<u>1.58401942s</u>

Joining across tables

As alluded to previously, the fact that Neo4j does not have to join tables together at query time means it can benefit from tremendous speed-ups relative to a Postgres implementation. Query 6 is almost 3 orders of magnitude faster than its Postgres counterpart, while Query 7 is a whopping 7x faster! The queries also read better and are less confusing than using joins in SQL.

- 6) Get the names of the Airlines operating flights from Chicago to Los Angeles

Cypher:

```
MATCH (ord:Airport {iata: 'ORD'})-[:HAS_DEPARTURE]->
      (f1:Flight)-[:FLIES_TO]->(lax:Airport {iata: 'LAX'}),
      (f1)-[:OPERATED_BY]->(al:Airline)
RETURN DISTINCT al.name
```

Postgres SQL:

```
SELECT name from airlines
INNER JOIN flights ON (airlines.iata = flights.airline)
WHERE origin_airport = 'ORD' AND destination_airport = 'LAX'
AND destination_airport = 'CLT'
```

Attempt	1	2	3	Avg
<i>Neo4j</i>	0.76818610s	0.81222974s	0.6413182s	<u>0.75028301s</u>
<i>PostgreSQL</i>	1.97668738s	1.57472943s	3.3357016s	<u>2.21939274s</u>

- 7) Get the names of destination airports from all flights originating in Wyoming

Cypher:

```
MATCH (hi:Airport {state: 'WY'})-[:HAS_DEPARTURE]->
      (f1:Flight)-[:FLIES_TO]->(ap:Airport)
RETURN DISTINCT ap.name
```

Postgres SQL:

```
SELECT name from airports
JOIN flights ON (airports.iata = flights.destination_airport)
WHERE flights.origin_airport IN
(SELECT iata from airports WHERE airports.state = 'WY')
```

Attempt	1	2	3	Avg
<i>Neo4j</i>	0.29222047s	0.34756797s	0.30003158s	<u>0.32918392s</u>
<i>PostgreSQL</i>	2.32982001s	2.14832110s	2.39080437s	<u>2.29873976s</u>

Bonus Query

This additional eighth query is presented only in Cypher as an illustration of Cypher's powerful ability to turn a problem statement written in English into an easily understandable query statement. A variation of such a query is likely used on travel booking and flight search websites like Google Flights to allow a customer to view all permutations of travel between two cities which include a layover stop. Because 200 iterations of this query run in approx 13 seconds, that means each query takes only about 0.07s to execute, which is fast enough to be used in web applications.

- 8) Get all possible flights from Alaska to Florida with a single layover in a different state of between 60 and 120 minutes

Cypher:


```

MATCH (ap1:Airport {state: 'AL'})-[:HAS_DEPARTURE]->
      (f1:Flight)-[:FLIES_TO]->(ap:Airport),
      (ap)-[:HAS_DEPARTURE]->(f12:Flight)-[:FLIES_TO]->
      (ap2:Airport {state: 'FL'})
WITH f1, f12, ap, ap1, ap2,
      toInteger(left(f1.scheduled_arrival, 2)) * 60 +
      toInteger(right(f1.scheduled_arrival, 2)) as f1_arrival,
      toInteger(left(f12.scheduled_departure, 2)) * 60 +
      toInteger(right(f12.scheduled_departure, 2)) as f12_departure
WHERE (f12_departure - f1_arrival) > 60 AND
      (f12_departure - f1_arrival) < 120
      AND ap.state <> 'FL' AND ap.state <> 'AL'
      AND f1.day = '1'
RETURN DISTINCT ap1.name, f1.scheduled_departure,
      f1.scheduled_arrival, ap.name,
      f12.scheduled_departure, f12.scheduled_arrival,
      ap2.name, (f12_departure - f1_arrival) as layover_time

```

Attempt	1	2	3	Avg
Neo4j	13.0806725s	12.99919647s	12.8740356s	<u>12.9283023s</u>

Flight departs from Birmingham-Shuttlesworth International Airport at 0900.
 Arrival at 1100 in Hartsfield-Jackson Atlanta International Airport
 65 minutes layover time.
 Next leg departs at 1205 and lands at 1322 in Daytona Beach International Airport

Flight departs from Birmingham-Shuttlesworth International Airport at 0900.
 Arrival at 1100 in Hartsfield-Jackson Atlanta International Airport
 74 minutes layover time.
 Next leg departs at 1214 and lands at 1326 in Gainesville Regional Airport

Flight departs from Birmingham-Shuttlesworth International Airport at 0900.
 Arrival at 1100 in Hartsfield-Jackson Atlanta International Airport
 85 minutes layover time.
 Next leg departs at 1225 and lands at 1350 in Orlando International Airport

Figure 5: Actual output sourced from Jupyter [notebook](#)

Conclusion

With Neo4j and other graph databases, developers are no longer constrained into using a query language that doesn't easily model their data domain. With Cypher, it is really easy to set up entities of related data and perform queries that take advantage of these relations. Indeed, in Neo4j, relationships are first-class citizens, and index-free adjacency means that these queries execute lightning-fast.

Appendices

Appendix 1

Match origin airport:

```
MATCH (f1:Flight),(ap:Airport)
WHERE f1.origin_airport = ap.iata
CREATE (ap)-[r:HAS_DEPARTURE]->(f1)
```

Match destination airport

```
MATCH (f1:Flight),(ap:Airport)
WHERE f1.destination_airport = ap.iata
CREATE (f1)-[:FLIES_TO]->(ap)
```

Match airline

```
MATCH (f1:Flight),(al:Airline)
WHERE f1.airline = al.iata
CREATE (f1)-[:OPERATED_BY]->(al)
```

Match tail

```
MATCH (f1:Flight),(tl:Tail)
WHERE f1.tail_number = tl.number
CREATE (f1)-[:USES]->(tl)
```