An Investigation of Neo4j Graph Platform with Timed Operations

Ashokkumar Mistry

UTM Continuing Education

Data Science

3252 Big Data Management Systems and Tools

Winter 2019

Abstract:

Traditional Relational Database Management Systems (RDBMS) databases have been used for software applications since the 1980s and has worked well with structured data that fits well into schemas of tables, columns, rows, and wherever queries are not carrying out join-intensive operations.

Much of the open-source community have taken the new database model of NoSQL and implemented various database types such as Document Stores, Graph Stores, Key-Value Stores, and Wide-Column Stores.

The Graph store, in particular, the Neo4j graph database will be explored in this paper. A small scale dataset will be implemented in Neo4J graph database using Cypher and the performance will be compared against a PostgreSQL database; with various timed operations presented - the results of which are not a benchmark test.

INTRODUCTION

Background

With the recent advancements in richly diverse content and connected data from e-commerce and social websites, the relational database has started to lose importance due to its reliance on strict schemas and expensive infrastructure. Organizations have come to the realization that RDBMS were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available today for BigData analytics.

Hence the rise of another type of database, optimized for connected data: the graph database. A graph database is an online database management system with Create, Read, Update and Delete (CRUD) operations working on a graph data model, that helps companies make sense of the masses of connected data that exist today. For instance, the early adopters of graph technology reimagined their businesses around the value of data relationships. These companies have now become industry leaders: LinkedIn, Google, and Facebook.

The biggest value that graphs bring to the development stack is their ability to store relationships and connections as first-class entities(1). It works particularly well when the relationships inside your data are important and your queries depend on exploring and exploiting them. In addition, the flexibility of a graph database model allows you to add new nodes and relationships without compromising your existing network or expensively migrating your data.

One of the most recent uses of graph databases was in the investigation of the highly publicized leaked documents known as the Panama Papers (2), which connected relationships of wealthy individuals to their offshore entities and the shell corporations used for illegal purposes, including fraud, kleptocracy, tax evasion, and evading international sanctions.

What is Neo4j?
Neo4j is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend for applications. The Neo4j database is also a property graph. You can add properties to nodes and relationships to further enrich the graph model. It is written in Java and Scala, and available for free download as a desktop version. Neo4j has also both a Community Edition and Enterprise Edition of the database. The Enterprise Edition includes everything that Community Edition has to offer, plus extra enterprise requirements such as backups, clustering, and failover abilities.

Modeling relational to a graph:
There are some similarities between a relational model and a graph model:

Relational	**Graph**
Rows	Nodes
Joins	Relationships
Table names	Labels
Columns	Properties

But, there are some ways in which the relational model differs from the graph model:

Relational	**Graph**
Each column must have a field value.	Nodes with the same label aren't required to have the same set of properties.
Joins are calculated at query time.	Relationships are stored on disk when they are created.
A row can belong to one table.	A node can have many labels.

Cypher Language:

Cypher is Neo4j's graph query language that allows users to store and retrieve data from the graph database. It is a declarative, SQL-inspired language for describing visual patterns in graphs using ASCII-Art syntax. It allows us to state what we want to select, insert, update, or delete from our graph data without a description of exactly how to do it. Through Cypher, users can construct expressive and efficient queries to handle needed create, read, update, and delete functionality.
Cypher query traversal navigates from starting node to related nodes finding quick responses with the assistance of multidimensional index (3).

Examples of cypher code:

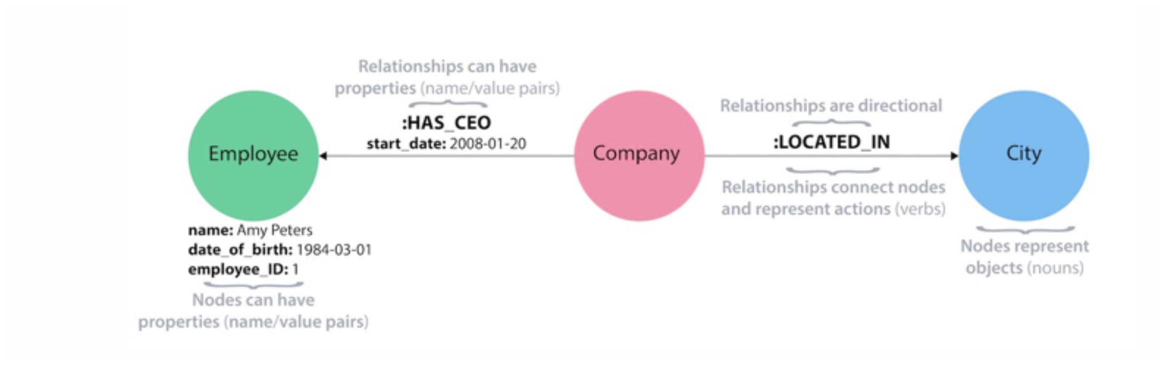

Relationships can have properties (name/value pairs)
:HAS_CEO
start_date: 2008-01-20
Employee
Company
Relationships are directional
:LOCATED_IN
City
Relationships connect nodes and represent actions (verbs)
name: Amy Peters
date_of_birth: 1984-03-01
employee_ID: 1
Nodes can have properties (name/value pairs)
Nodes represent objects (nouns)

Fig1: Diagram of the typical model design showing nodes and relationships

```
()                //anonymous node (no label or variable) can
refer to any node in the database
(c:Company)       //using variable c and label Company
(:Company)        //no variable, label Company
(work:Company)    //using variable work and label Company

//data stored with this direction
CREATE (c:Company)-[:HAS_CEO]->(e:Employee)

//query relationship backwards will not return results
MATCH (c:Company)<-[:HAS_CEO]-(e:Employee)

//better to query with undirected relationship unless sure of
direction
MATCH (c:Company)-[:HAS_CEO]-(e:Employee)

(c:Company {name: “Bell Canada”})-[rel:HAS_CEO]->(e:Employee
{name: “Amy Peters"})

MATCH (e:Employee)
RETURN e
```

Cypher Keywords

There are a few words in Cypher reserved for specific actions in parts of a query. The **MATCH** keyword is similar to **SELECT** in SQL, it is what searches for an existing node, relationship, label, property, or pattern in the database. The **RETURN** keyword specifies what values or results you might want to return from a Cypher query.

PROCEDURE:

To investigate the power of graph databases, Neo4j Community Edition was implemented on a MacBook Air running MAC OS on a single Intel Core i5 1.8 GHz processor with 2 cores, and 8GB memory.

The dataset used is the UK Road Safety consisting of accidents and the vehicles from 2005 to 2016 obtained from Kaggle. The original data comes from the Open Data website of the UK government, where they have been published by the Department of Transport, and consists of two relatively large files **Accident_Information.csv** (672.77 MB) and **Vehicle_Information.csv** (614.57 MB). Before bringing the data into Neo4j the CSV files were imported into a PostgreSQL database for analysis and further comparison with SQL against graph query operations.

Before beginning any data import to Neo4j it helps to diagram the property graph model. The diagram tool to do this can be pencil and paper, a flip-chart or whiteboard. Instead, I used a simple Javascript tool called Arrows, that allow us to define the model in a web browser.

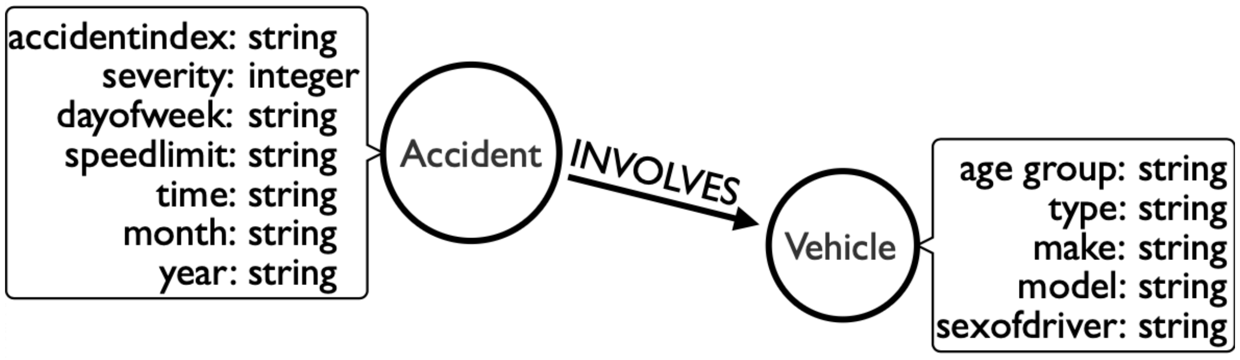


Fig 2: Arrow Tool's graph design for the Accident and Vehicle dataset

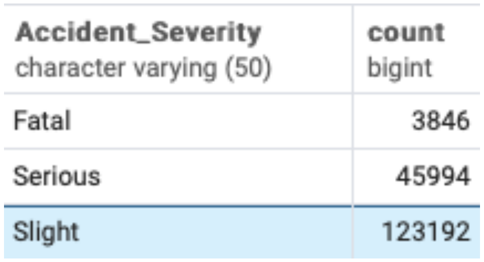
The data is imported into Neo4j, the steps carried out are in the Appendix.

Create, Read, Update, and Delete (CRUD) operations were carried out with Neo4j and PostgreSQL and the results included in the appendix.

One such result is included here: We will find all accidents that involved a motorcycle and compute the count of the severity of the accident.

The CRUD commands for Neo4J can be run from a Cypher shell or from a browser interface at http://localhost:7474 which become available when the Neo4J service is started.
PostgreSQL:

```
SELECT "Accident_Severity", COUNT("Accident_Severity")
FROM accident
INNER JOIN Vehicle ON accident."Accident_Index" = vehicle."Accident_Index"
WHERE "Vehicle_Type" LIKE 'Motorcycle%'
GROUP BY “Accident_Severity"
```

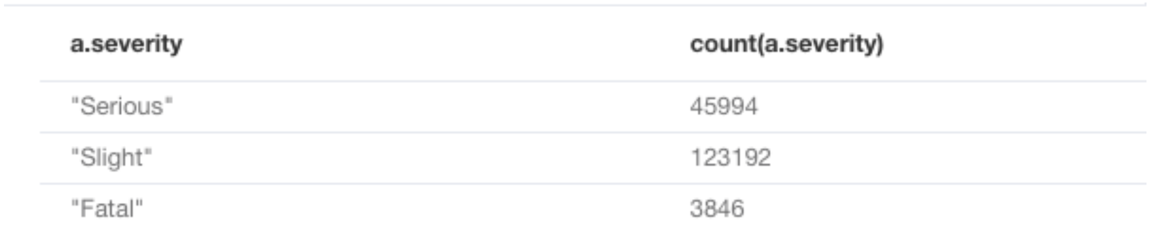


Accident_Severity character varying (50)	count bigint
Fatal	3846
Serious	45994
Slight	123192

Successfully run. Total query runtime: 13 secs 548 msec. 3 rows affected.

Neo4J:

```
MATCH (a:Accident)-[:INVOLVES]->(v:Vehicle)
WHERE v.vehicletype CONTAINS 'Motorcycle'
RETURN a.severity, count(a.severity);
```



a.severity	count(a.severity)
"Serious"	45994
"Slight"	123192
"Fatal"	3846

Started streaming 3 records after 2173 ms and completed after 2174 ms.

CONCLUSION

During implementation, it was found that Neo4J invokes a JVM that increases in memory usage as more nodes and relationships are added. This meant that the machine has to have sufficient memory for a large graph database.

The performance of certain CRUD operations does not appear to favour the graph database. The cypher queries ran faster on read operations but were slower on delete and update. The create operation can require more effort in establishing the nodes and relationships for large implementations, and so take longer to code and run.

Even with the performance limits, as demonstrated in this report with a simple use-case, I can see there are tangible benefits, and which show the considerable potential of graph databases, especially when there are many layers of connected information.

Appendix:

Details of the Dataset

Accident_Information.csv : Each line represents a single traffic accident (identified by the Accident_Index column) and its various properties.

“Accident_Index","1st_Road_Class","1st_Road_Number","2nd_Road_Class","2nd_Road_Number","Accident_Severity","Carriageway_Hazards","Date","Day_of_Week","Did_Police_Officer_Attend_Scene_of_Accident","Junction_Control","Junction_Detail","Latitude","Light_Conditions","Local_Authority_(District)","Local_Authority_(Highway)","Location_Easting_OSGR","Location_Northing_OSGR","Longitude","LSOA_of_Accident_Location","Number_of_Casualties","Number_of_Vehicles","Pedestrian_Crossing-Human_Control","Pedestrian_Crossing-Physical_Facilities","Police_Force","Road_Surface_Conditions","Road_Type","Special_Conditions_at_Site","Speed_limit","Time","Urban_or_Rural_Area","Weather_Conditions","Year","InScotland"

"200501BS00001","A","3218",NA,"0","Serious","None", 2005-01-04,"Tuesday","1","Data missing or out of range","Not at junction or within 20 metres",51.489096,"Daylight","Kensington and Chelsea","Kensington and Chelsea","525680","178240",-0.19117,"E01002849",1,1,"0","1","Metropolitan Police","Wet or damp","Single carriageway","None",30,"17:42","Urban","Raining no high winds”,2005,”No”

(“2047256" rows, “34” columns)

Vehicle_Information.csv : Each line represents the involvement of a unique vehicle in

an accident, featuring various properties regarding the vehicle itself and its passengers.

“Accident_Index","Age_Band_of_Driver","Age_of_Vehicle","Driver_Home_Area_Type","Driver_IMD_Decile","Engine_Capacity_.CC.","Hit_Object_in_Carriageway","Hit_Object_off_Carriageway","Journey_Purpose_of_Driver","Junction_Location","make","model","Propulsion_Code","Sex_of_Driver","Skidding_and_Overturning","Towing_and_Articulation","Vehicle_Leaving_Carriageway","Vehicle_Location.Restricted_Lane","Vehicle_Manoeuvre","Vehicle_Reference","Vehicle_Type","Was_Vehicle_Left_Hand_Drive","X1st_Point_of_Impact","Year"

"200401BS00001","26 - 35",3,"Urban area","4",1588,"None","None","Data missing or out of range","Data missing or out of range","ROVER","45 CLASSIC 16V","Petrol","Male","None","No tow/articulation","Did not leave carriageway","0","Going ahead other","2","109","Data missing or out of range”,”Front", 2004

("2177205" rows, "24" columns)

Data import into Neo4j

Here are the steps carried out:

The AccidentIndex property for the accident label uniquely identifies the accident, so we will need to create a unique constraint on this property.

```
CREATE CONSTRAINT ON (a:Accident) ASSERT a.accidentindex IS UNIQUE
```

Adding Accident Node:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///Accident_Information.csv" AS line
WITH line, SPLIT(line.Date,'-') AS date
CREATE (a:Accident {accidentindex: line.Accident_Index})
SET a.severity = line.Accident_Severity
SET a.dayofweek = line.Day_of_Week
SET a.speedlimit = line.Speed_limit
SET a.time = line.Time
SET a.month = TOINT(date[1])
SET a.year = TOINT(date[0])
```

Adding Vehicle Node:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///Vehicle_Information.csv" AS line
MATCH (a:Accident {accidentindex: line.Accident_Index})
CREATE (v:Vehicle)
SET v.vehicletype = line.Vehicle_Type
SET v.make= line.make
SET v.model = line.model
SET v.sexofdriver = line.Sex_of_Driver
SET v.agegroup = line.Age_Band_of_Driver
CREATE (a)-[:INVOLVES]->(v)
```

Timed Operations:

PostgreSql:

```
SELECT "Accident_Severity"
FROM accident
INNER JOIN vehicle_
ON accident."Accident_Index" = vehicle."Accident_Index"
WHERE "Sex_of_Driver" = 'Male';
```

Successfully run. Total query runtime: 4 secs 445 msec. 1383012 rows affected.

Neo4j:
```
MATCH (a:Accident)-[:INVOLVES]->(v:Vehicle)
WHERE v.sexofdriver = 'Male'
RETURN a.severity
```
Started streaming 1383012 records after 1 ms and completed after 10396 ms, displaying first 1000 rows.

UPDATE:
PostGreSQL
```
UPDATE accident
SET "Day_of_Week" = 'Wednesday'
WHERE "Accident_Index" = ‘200501BS00014'
```
UPDATE 1, Query returned successfully in 1 secs 317 msec.

Neo4J:
```
MATCH (n)
WHERE n.accidentindex = "200501BS00014"
SET n.dayofweek = “Wednesday"
```
Set 1 property, completed after 6010 ms

DELETE:
PostgreSQL:

```
DELETE FROM accident
WHERE "Accident_Index" = ‘2017984122617'
```
Query returned successfully in 863 msec.

Neo4J:
```
MATCH (accident { accidentindex: '2017984122617' })
DETACH DELETE accident
```
Deleted 1 node, completed after 5135 ms.

REFERENCES:

1. Guest View: Relational vs. graph databases: Which to use and when? https://sdtimes.com/databases/guest-view-relational-vs-graph-databases-use/

2. Why Panama Papers Journalists Use Graph Databases. https://www.forbes.com/sites/metabrown/2016/04/30/why-panama-papers-journalists-use-graph-databases/#6ce2fb417467

3. An Efficient Index based Query handling model for Neo4j: Anita Brigit Mathew, S. D. Madhu Kumar