Neo4J Lab (B)

UK Vehicle Accident Database

Overview:

In this lab you will load some traffic accident data for the UK and carry out some basic analysis.
First you will need to download and install Neo4J, and then import the data to carry out some analysis.

Installation

Go to www.macs.hw.ac.uk/~pb56/neo4j.tar.gz and download the file to your machine.

Open a linux terminal and find your downloaded file (it will probably be in your Downloads folder):

```
cd ~/Downloads
ls
```

If you do not see the file you downloaded ask for help!

Extract the file using tar:
You will need to change the name of the file from 3.4.4 to reflect the file you downloaded!

```
tar -zxf neo4j-community-3.4.4-unix.tar.gz
```

Look at the contents of the Downloads directory, and enter that directory:

```
ls
cd neo4j-community-3.4.4
```

Again, look at the contents of the directory and this time enter the bin subdirectory:

```
ls
cd bin
```

Look at the contents of this directory too:

```
ls
```

You will see a range of programs. To start Neo4j type:

```
./neo4j start
```

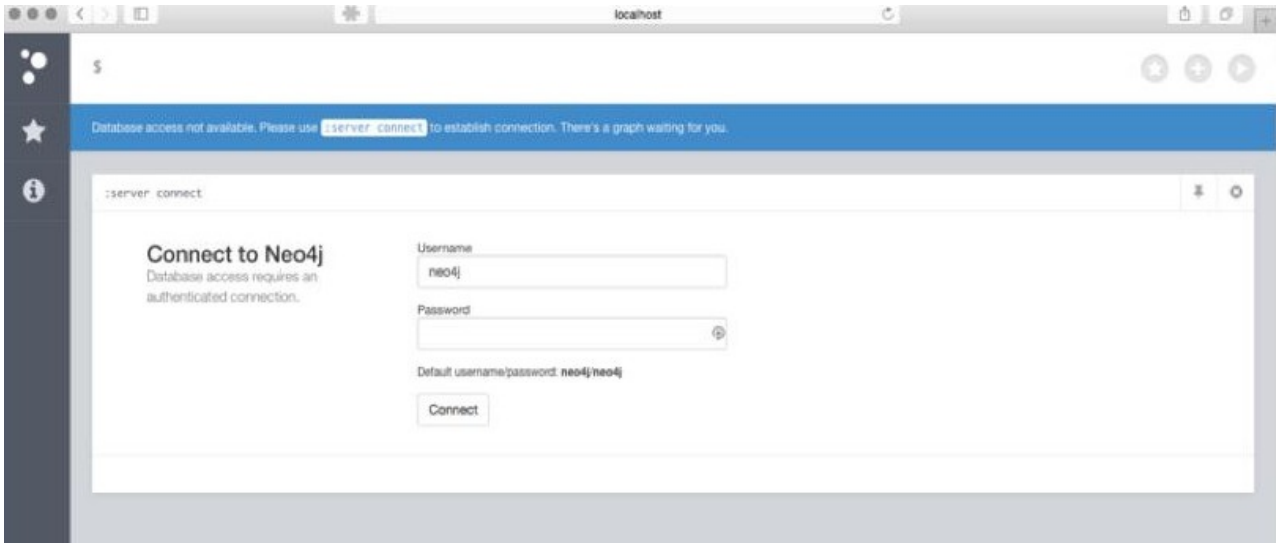
NOTE :
When you finish the lab you need to shutdown Neo4j, return to this terminal window and type:

```
./neo4j stop
```

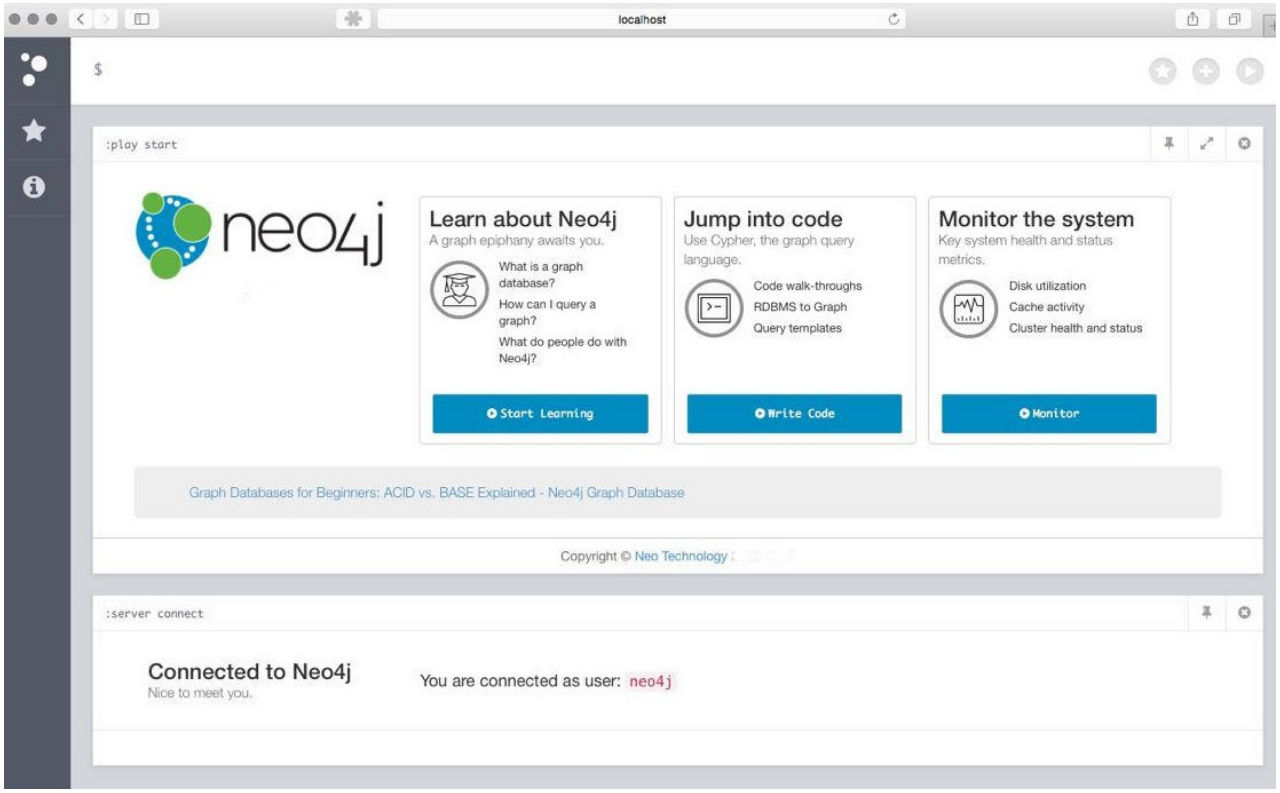
Neo4J has a “localhost exception”, which means the local user (you) has admin access.
In your web browser go to:

```
http://localhost:7474/browser/
```

You will see:



Enter the default username (neo4j) and password (neo4j). When asked to enter a new password choose **graphdb** . Now you will see:



Using Neo4J

Stage 1: Check the CSV files using Neo4J before loading them into a Database

Now download the CSV files we'll use in this lab from:
www.macs.hw.ac.uk/~pb56/accidents.tar.gz

Neo4J by default limits importing data to the IMPORT folder.
For this version it can be found in your install directory here: **/neo4j-commmunity-3.4.4/import**

On the VM this is located in /var/lib/neo4j/import

- unzip the file and copy the 3 csv files to the import folder

(Linux cmd to unzip: tar -xf accidentx.tar.gz)

From the Neo4J web browser page carry out the following

Check the data held for each accident:

```
LOAD CSV WITH HEADERS FROM "file:///accidents.csv" as row RETURN row LIMIT 5
```

then look at the vehicles involved in the crashes:

```
LOAD CSV WITH HEADERS FROM "file:///vehicles.csv" as row RETURN row LIMIT 5
```

and the vehicle types:

```
LOAD CSV WITH HEADERS FROM "file:///vehicle_type.csv" as row RETURN row LIMIT 5
```

Stage 2: Diagram the Graph Data Model

This is OPTIONAL - to help you build a GRAPH diagram and understand the dataset.

In this case we'll use 1 entity type for each of the CSV files.
Use this webtool to create 3 entitys (accident; vehicle; vehicle type) http://www.apcjones.com/arrows/

- double click on the starting node and name it 'accident'
- click the edge of the circle to add a linked circle - name it 'vehicle'
- click on the vehicle cirlce to add a 3rd circle and name it 'vehicle_type'

Now you have a basic graph data model, but need to add the relationship details.
- double click on the link between accident and vehicle and name it 'involves'
- double click on the link between vehicle and vehicle_type and name it 'is_type'

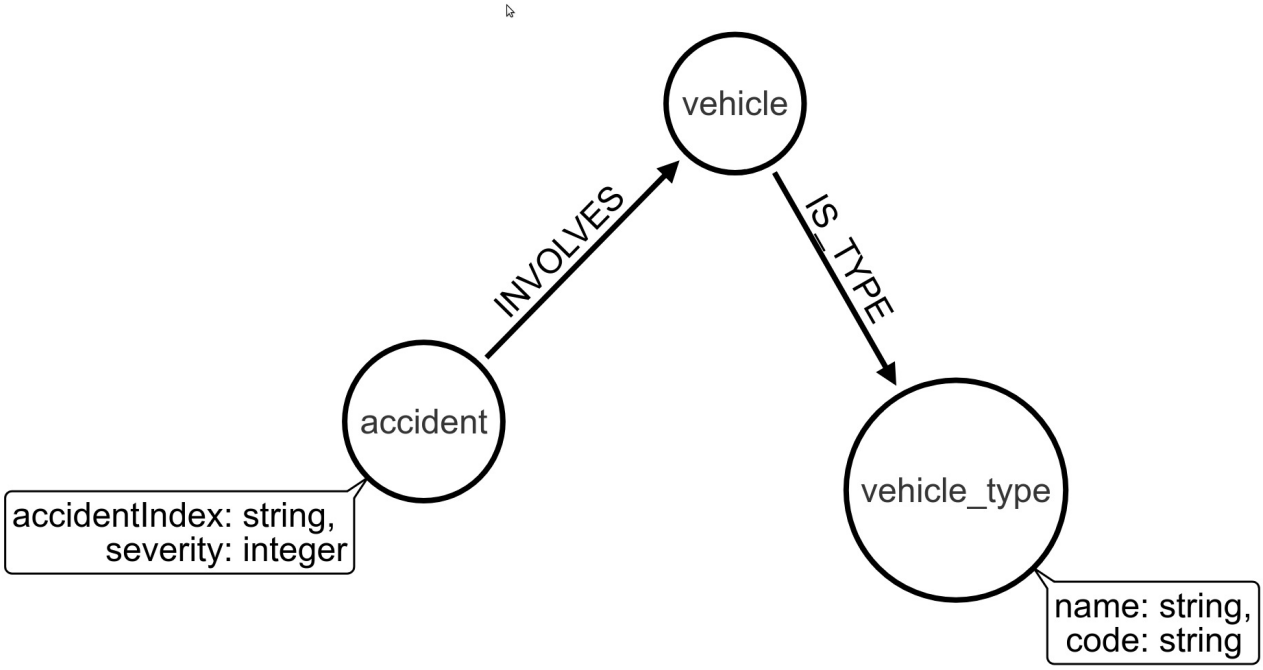
Finally add some of the properties we'll be using. For the accident node add the following details:

```
accidentIndex: string,
severity: integer
```

.. and for the vehicle_type node add the following properties:

```
name: string,
code: string
```

Your final graph model should look something like this:



Stage 3: Creating the Database

Back in the neo4J browser window let's create this graph database.

The **accidentIndex** in the accident csv is a unique value which can be used to link to the vehicle csv, to find which vehicles were involved in any accident. We therefore need to create a constraint in our Neo4J database to ensure that the values are unique.

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

Next let's load the accident data using mutiple transactions (period committ), and to avoid duplicates we'll use the MERGE command (without the constraint created above this would be very slow), and cast the severity to Integers:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///accidents.csv" AS row
```

```
MERGE (a:Accident {accidentIndex: row.Accident_Index})
ON CREATE SET a.severity = toInteger(row.Accident_Severity)
```

This should add 129982 labels, created 129982 nodes, set 259964 properties.

Now we'll load the vehicle data - this time we want to match the **accidentIndex** in each record to that in the accident node by using the **MATCH** function. Also we'll import all of the properties from the csv using **SET v+=row**, and create the **INVOLVES** relationship between the accident and vehicle nodes:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///vehicles.csv" AS row
MATCH (a:Accident {accidentIndex: row.Accident_Index})
CREATE (v:Vehicle)
SET v+= row
CREATE (a)-[:INVOLVES]->(v)
```

This should result in the addition of 238926 labels, created 238926 nodes, set 5495298 properties, created 238926 relationships.

Finally we'll load the vehicle_type data. It's a good idea to use MERGE to ensure a unique list of vehicle code types are loaded. For each row in the csv we'll set the vehicle type name (was called label in the csv), and link the vehicle types to the vehicle node. Then we'll add the relationship between vehicles and vehicle_types.

```
LOAD CSV WITH HEADERS FROM "file:///vehicle_type.csv" AS row
MERGE (t:VehicleType {code: row.code })
SET t.name=row.label
WITH t,row
MATCH (v:Vehicle) WHERE v.Vehicle_Type = row.code
CREATE (v)-[:IS_TYPE]->(t)
```

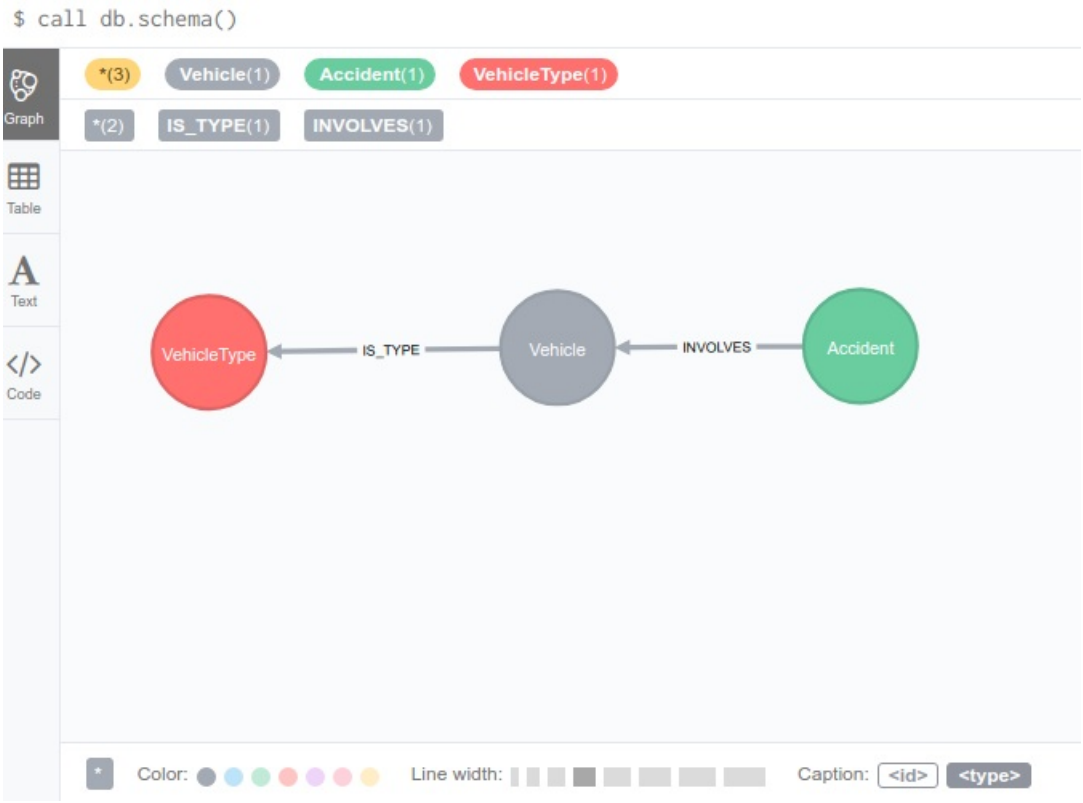
This should create 21 labels, created 21 nodes, set 42 properties, created 238926 relationships.

So every vehicle (238926) is now connected to a vehicle_type.

Let's check the schema to see if it looks like our intended design:

```
    call db.schema()
```

it should look something like this:



- You can change the label and line thickness by selecting a node/relationship from the list above the graph diagram, then use the options under the diagram to make changes.

Stage 3: Query the Data using Cypher

1a) How many types of motorcycle are there in the vehicle type dataset?

```
MATCH (t:VehicleType)
WHERE t.name CONTAINS "Motorcycle"
RETURN t.name
```

1b) This time forcing the check to be in lower case

```
MATCH (t:VehicleType)
WHERE lower(t.name) CONTAINS "motorcycle"
RETURN t.name
```

- What difference did you notice in the results 1a and 1b ? _____

2a) How many vehicles were involved in accidents where the driver was less than 20 years old?

```
MATCH (v:Vehicle)
WHERE v.Age_of_Driver < 20
RETURN count(v)
```

Does this seem OK?
Try the above again but use the toInteger() function (see syntax in 2b below). What is the result now? ____

2b) Add a new property to store integer age values

```
MATCH (v:Vehicle)
SET v.age =  toInteger(v.Age_of_Driver)
```

Now try the query above to find how many drivers were less than 20yrs old using this property.

```
MATCH (v:Vehicle)
WHERE v.age < 20
RETURN count(v)
```

Note how long that took to run: _____ ms

Now add an index to the age property

```
CREATE INDEX ON :Vehicle(age)
```

..and run that query again… how long did it take to run this time? _____ ms

3) Count how many vehicles were involved in accidents for people aged 20, 30, 40

Rather than using v.age=20 OR v.age=30 OR v.age=40 we can use the IN [] operator.
Neo4J will automatically group the data if the RETURN includes the age property as follows:

```
MATCH (v:Vehicle)
WHERE v.age in [20,30,40]
RETURN count(v),v.age
ORDER by v.age
```

How many vehicles were in accidents with a driver age 20:__ age 30:_ age 40:_ ?

To find the highest incidents for these 3 ages we can use ORDER BY and LIMIT as follows:

```
MATCH (v:Vehicle)
WHERE v.age in [20,30,40]
RETURN count(v) as c ,v.age
ORDER by c DESC
LIMIT 1
```

4a) Create a list of accidents that involve a motorcycle

```
MATCH (a:Accident)-[:INVOLVES]->(v:Vehicle)-[:IS_TYPE]->(t:VehicleType)
WHERE t.name CONTAINS "Motorcycle"
RETURN distinct a.accidentIndex,a.severity
```

4b) What is the average (mean) severity for accidents containing a motorcycle?

```
MATCH (a:Accident)-[:INVOLVES]->(v:Vehicle)-[:IS_TYPE]->(t:VehicleType)
WHERE t.name CONTAINS "Motorcycle"
WITH Distinct a.accidentIndex as accid, a.severity as sev
RETURN  avg(sev)
```

5) What is the average (mean) severity for accidents containing a motorcycle for each type of motorcycle (i.e. like a GROUP BY)?

```
MATCH (a:Accident)-[:INVOLVES]->(v:Vehicle)-[:IS_TYPE]->(t:VehicleType)
WHERE t.name CONTAINS "Motorcycle"
WITH Distinct a.accidentIndex as accid, a.severity as sev, t.name as name
RETURN name, avg(sev) as avg_severity
ORDER BY avg_severity
```

6) This time we'll include a count and electric motorcyles.

```
MATCH (a:Accident)-[:INVOLVES]->(v:Vehicle)-[:IS_TYPE]->(t:VehicleType)
WHERE lower(t.name) CONTAINS "motorcycle"
WITH Distinct a.accidentIndex as accid, a.severity as sev, t.name as name
RETURN name, avg(sev) as avg_severity
ORDER BY avg_severity
```

7) How many accidents involved more than 8 vehicles?

```
MATCH (a:Accident) -[:INVOLVES]-> (v:Vehicle)
WITH a,count(v) as c WHERE c > 8
RETURN count(DISTINCT a)
```

8a) How many accidents involved a Car, how many involved a Tram, and how many involved a Car and a Tram?

```
MATCH (a:Accident) -[:INVOLVES]-> (v1:Vehicle)-[:IS_TYPE]->(vt1:VehicleType)
WHERE  vt1.name='Car'
RETURN count (DISTINCT a)



MATCH (a:Accident) -[:INVOLVES]-> (v1:Vehicle)-[:IS_TYPE]->(vt1:VehicleType)
WHERE  vt1.name='Tram'
RETURN count (DISTINCT a)



MATCH  (vt1:VehicleType) <-[:IS_TYPE]-(v1:Vehicle) < - [:INVOLVES] - (a:Accident) -[:INVOLVES]->
(v2:Vehicle)-[:IS_TYPE]->(vt2:VehicleType)
WHERE  vt1.name='Tram' and vt2.name='Car'
RETURN count (DISTINCT a)
```

Note: You could also write this query as follows:

```
MATCH (a:Accident) -[:INVOLVES]-> (v1:Vehicle)-[:IS_TYPE]->(vt1:VehicleType)
MATCH (a:Accident) -[:INVOLVES]-> (v2:Vehicle)-[:IS_TYPE]->(vt2:VehicleType)
WHERE  vt1.name='Tram' and vt2.name='Car'
RETURN count (DISTINCT a)
```

8b) How many accidents involved a Car and Tram where the car driver was less than 20 years old?

```
__________
```

Notes:

If you wish to delete all data in the Neo4J database:

```
MATCH (n)
OPTIONAL MATCH (n)-[r]-()
DELETE n,r
```

Alternatively shut down the Neo4J server and run this command:

```
rm data/databases/graph.db
```