

## INSTALLATION

Go to <http://neo4j.com/download/> and download the **community** edition.

On the linux command line find your downloaded file and extract it:

```
tar -zxvf neo4j-community-2.2.2-unix.tar.gz
```

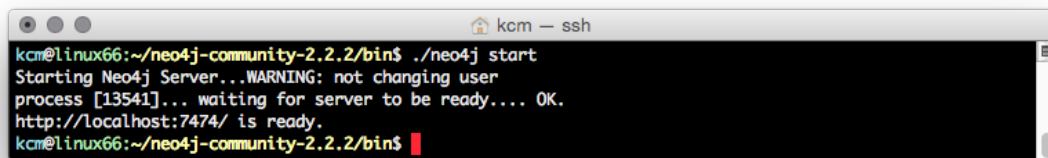
To run neo4j, enter the neo4j sub-directory called “bin”:

```
cd neo4j-community-2.2.2/bin
```

Now start the database:

```
./neo4j start
```

Your terminal window should like:

A terminal window titled 'kcm - ssh' showing the execution of the './neo4j start' command. The output indicates the Neo4j Server is starting with a warning about not changing the user process [13541], and that the server is ready at http://localhost:7474/.

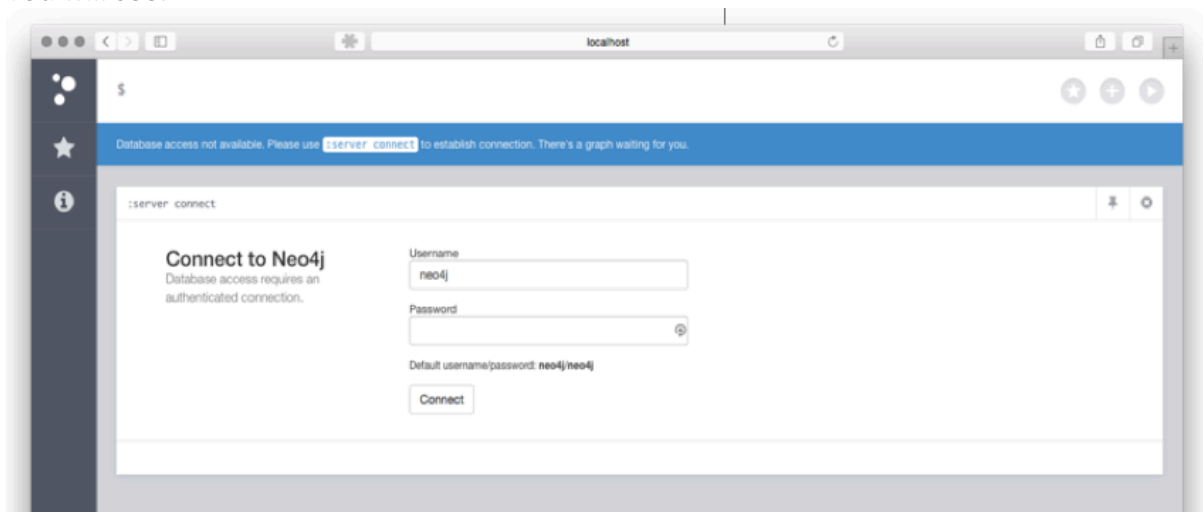
```
kcm@linux66:~/neo4j-community-2.2.2/bin$ ./neo4j start
Starting Neo4j Server...WARNING: not changing user
process [13541]... waiting for server to be ready... OK.
http://localhost:7474/ is ready.
kcm@linux66:~/neo4j-community-2.2.2/bin$
```

By default Neo4J has a “localhost exception”, which means the local user (you) has admin access.

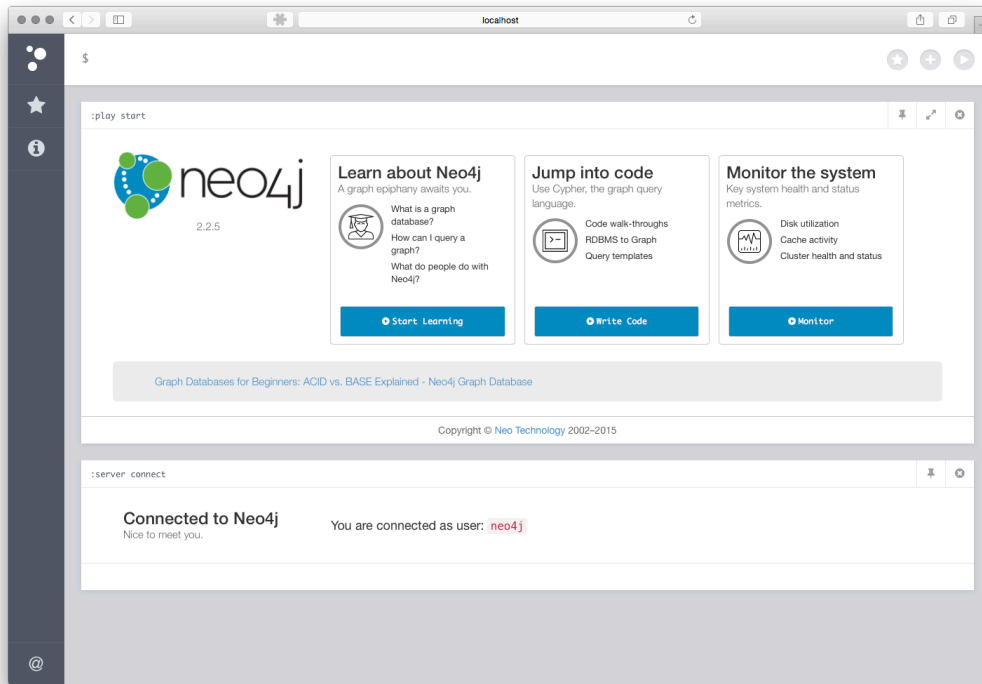
In your web browser go to:

<http://localhost:7474/browser/>

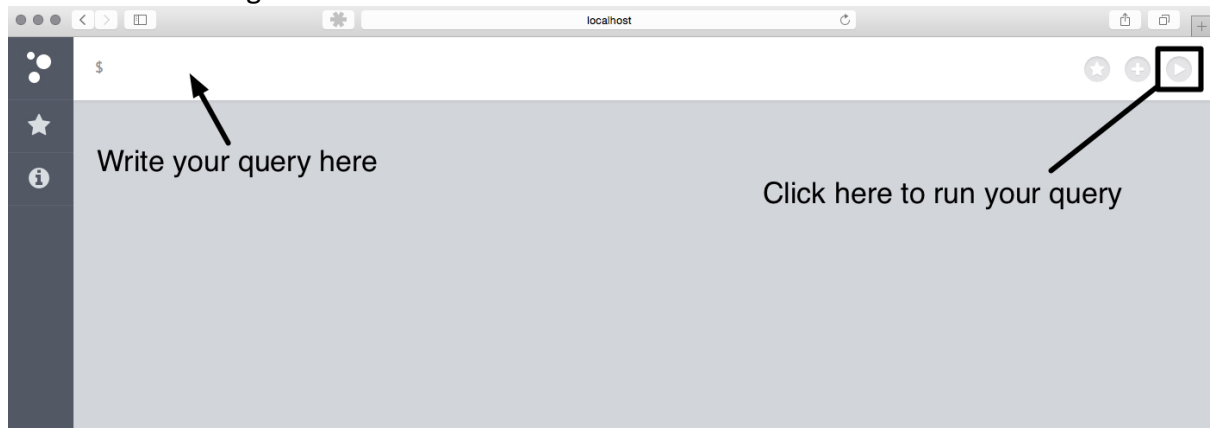
You will see:



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



Close the middle and bottom tiles (use the cross in top right corner of each tile). You will be left with something like:

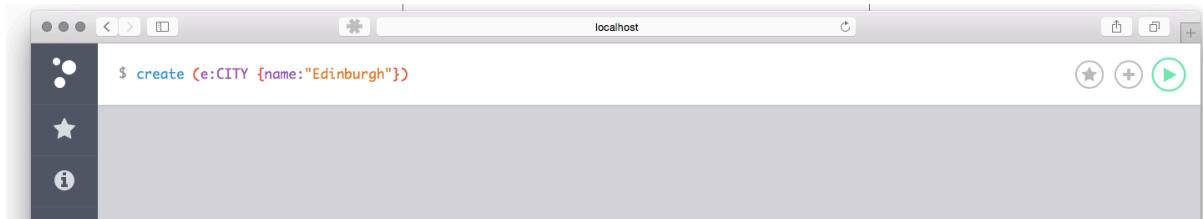


## BASICS

Our graph is going to have UK cities as nodes, and modes of transport between them as relationships.

Our first node is Edinburgh, notice we assign the node the role of CITY:  
create (e:CITY {name:"Edinburgh"})

Hint: to run the query click the button highlighted in green:



Task: Add a node for London.

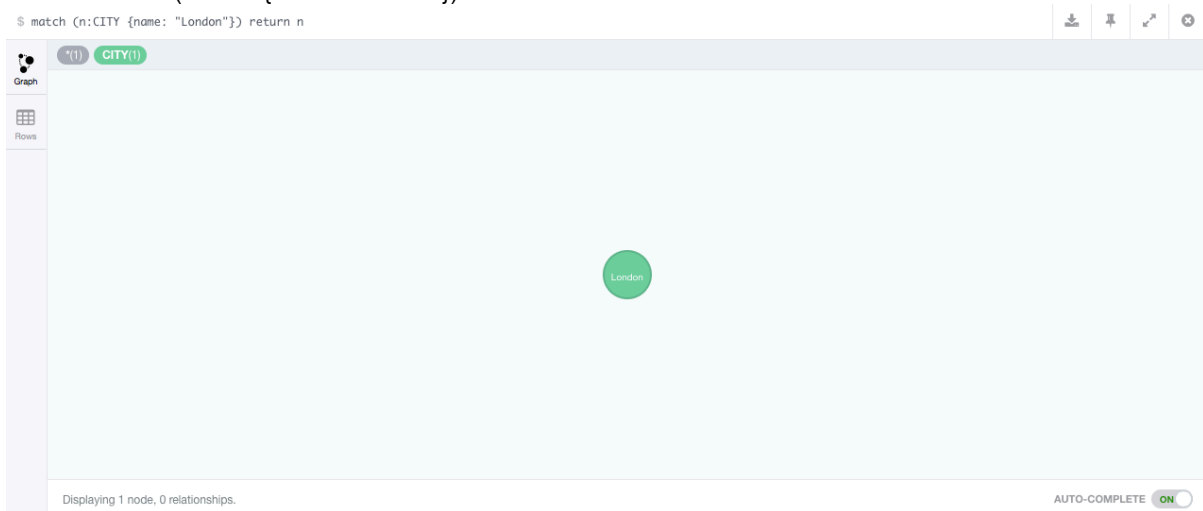
To find the list of all nodes:

`match (n) return n`

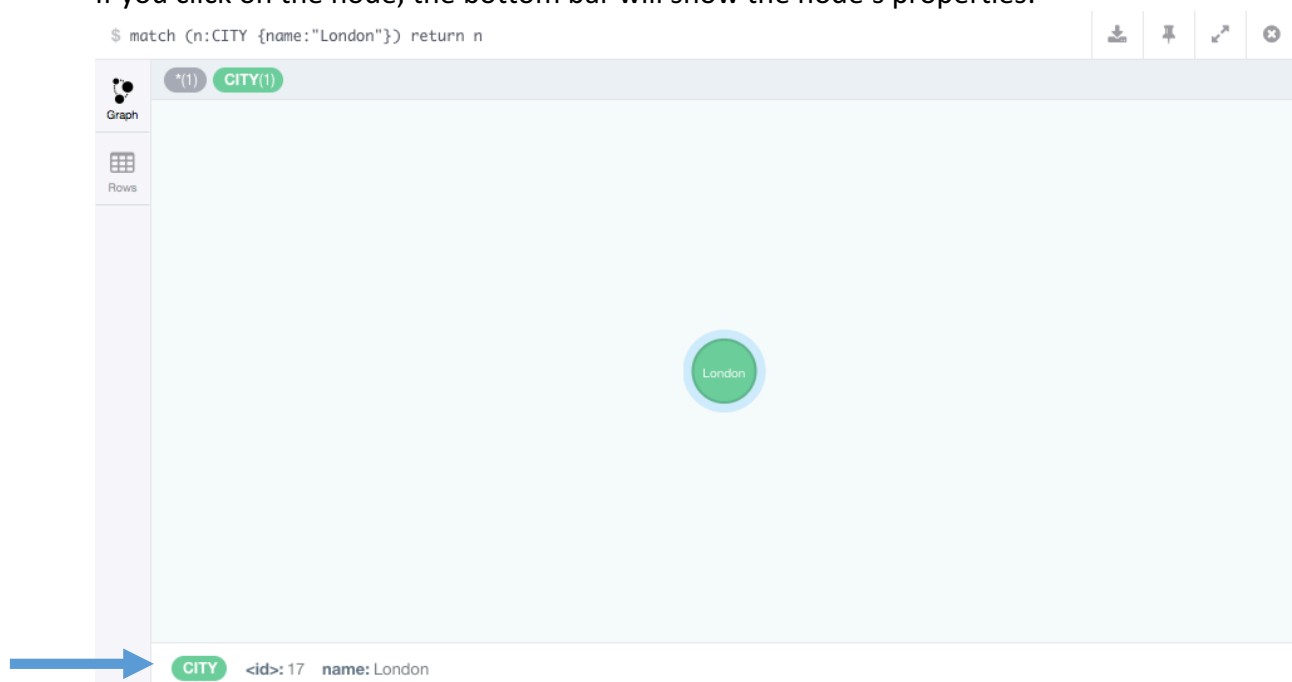
Hint: use the “Rows” view to see your results.

To query for the London node:

`match (n:CITY {name: "London"}) return n`



If you click on the node, the bottom bar will show the node’s properties:



Our first relationship will indicate that it is possible to drive from Edinburgh to London in 7 hours. This will use a combination of a *match* query (to find the nodes) and a *create* query (to link the nodes):

```
match (l:CITY {name:"London"}), (e:CITY {name:"Edinburgh"}) create (e)-[:DRIVE {time: 7}]->(l)
```

Task: add the following relationships between London and Edinburgh:

TRAIN, 5 hours

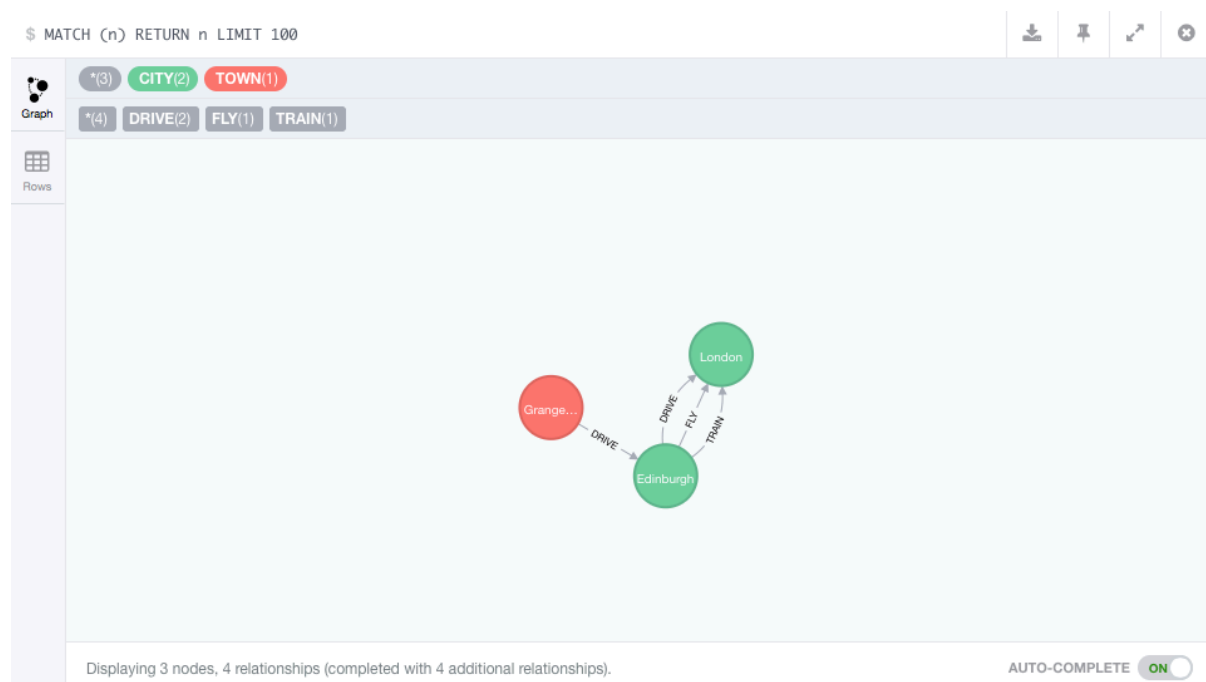
FLY, 1 hour

Task: Add the town of Grangemouth, give it a TOWN role. Link Grangemouth to Edinburgh via a CAR relationship where the name is *m9*.

```
match (g:TOWN {name:"Grangemouth"}), (e:CITY {name:"Edinburgh"}) create (g)-[:DRIVE {time: 1}]->(e)
```

Run the following query to inspect your graph:

```
match (n) return n
```



When you look at the full graph, notice that Grangemouth is a different colour. This represents the different role it has.

Edinburgh's population is 492, 680. To add that to the Edinburgh node we must first *match* the Edinburgh node and then *set* the population value:

```
match (n:CITY {name:"Edinburgh"}) set n.population=492680 return n
```

Task: For the London node, add the following information:

Population 8630000

Size 1,572

Founded 43AD

You can compare the property values within a graph. For example, to determine if the population of Edinburgh is larger than London:

```
match (e:CITY{name:"Edinburgh"}), (l:CITY {name:"London"}) return e.population >= l.population
```

To delete a property from a node, *match* the node and then *remove* the property:

```
match (n:CITY {name:"Edinburgh"}) remove n.population return n
```

To delete all nodes and relationships from your graph run the query:

```
match (n) optional match (n)-[r]-() delete n,r
```

If the contents of the *match* clause are not found the query will fail. If the contents of the *optional match* are found they will be deleted; however, if the *optional match* is not found the query will still work.

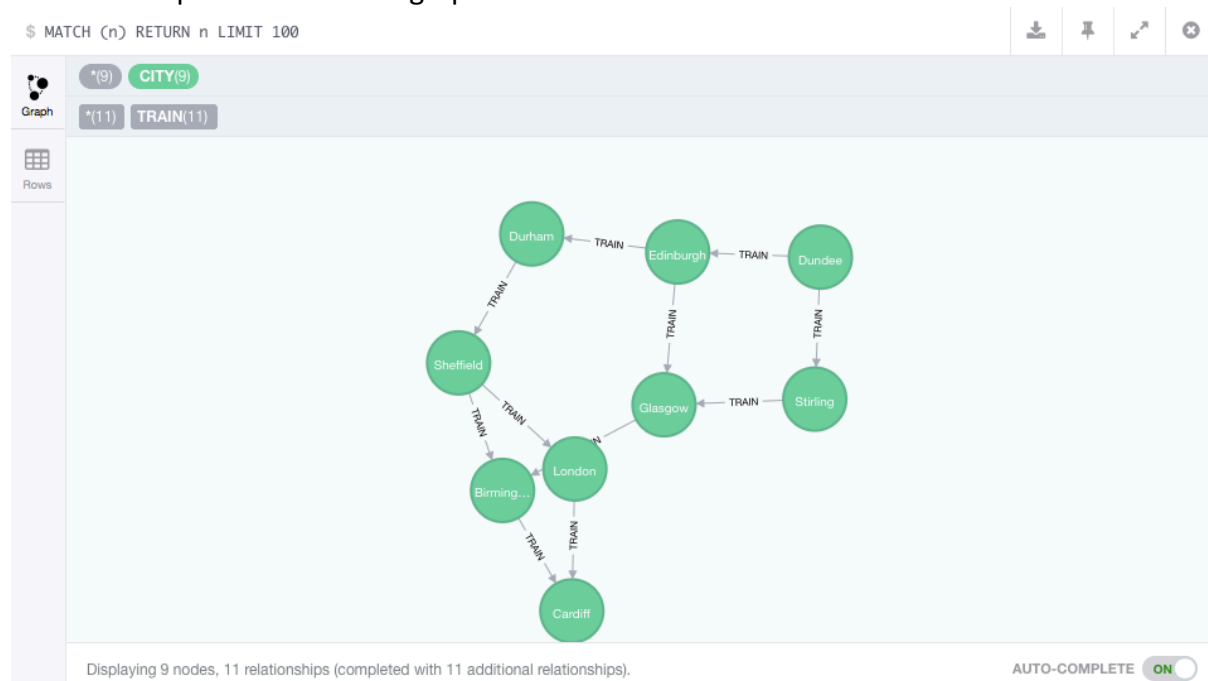
## A BIGGER GRAPH

Copy the content of XXXXX

And paste it into the query box at the top of your Neo4j window. Run the query. Your output should be similar to:

Added 9 labels, created 9 nodes, set 9 properties, created 11 relationships, statement executed in 50 ms.

Your first step is to look at the graph:



Note: You can drag the nodes around to rearrange the graph, which might make it easier to read.

It contains pseudo information regarding trains journeys in the UK. We have a number of cities and relationships which indicate if you can travel between the cities on the train.

To list all the destinations you can reach with a single leg journey from Edinburgh:

```
match (n:CITY {name:"Edinburgh"})--(d:CITY) return d.name
```

Task: List all the destinations you can reach with a multiple leg journey from Dundee

Hint: Define the relationship as \*

Hint 2: To ensure you only have unique answers change the return to "return distinct ... "

```
match p = (n:CITY {name:"Dundee"})-[*]->(d:CITY {name:"Cardiff"}) return d.name
```

To list all the routes from Sheffield to Cardiff we create a variable (*p*) in which we store all the paths between the two nodes:

```
match p = (n:CITY {name:"Sheffield"})-[:TRAIN*]->(d:CITY {name:"Cardiff"}) return p
```

To list the routes and their length append length(*p*) to the return clause:

The screenshot shows a Cypher query interface with the following query: `$ match p = (n:CITY {name:"Sheffield"})-[:TRAIN*]->(d:CITY {name:"Cardiff"}) return p, length(p)`. The results are displayed in a table with two columns: *p* and *length(p)*. The first row shows a path *p* as a list of nodes: `[{name: 'Sheffield'}, {name: 'east coast'}, {name: 'London'}, {name: 'great western'}, {name: 'Cardiff'}]` and its length as 2. The interface also includes a 'Graph' view icon, a 'Rows' view icon, and a status bar at the bottom indicating 'Returned 2 rows in 34 ms.'

<i>p</i>	<i>length(p)</i>
[{name: 'Sheffield'}, {name: 'east coast'}, {name: 'London'}, {name: 'great western'}, {name: 'Cardiff'}]	2

Returned 2 rows in 34 ms.

Notice that the path includes the name of the TRAIN relation between the cities (e.g., east coast).

Task: what is the length of the shortest path between Dundee and Cardiff?

Hint: Your *return* clause should use the *min()* function

```
match p = (n:CITY {name:"Dundee"})-[*]->(d:CITY {name:"Cardiff"}) return min(length(p))
```

Task: Create a FLY relationship between Edinburgh and Cardiff, with the relationship name *ba*. What is the length of the longest and shortest paths now?

```
match p = (n:CITY {name:"Dundee"})-[*]->(d:CITY {name:"Cardiff"}) return min(length(p)), max(length(p))
```

If we do not wish to fly, we should restrict our path query to the TRAIN relationship. We do this by specifying the relationship:

... [:TRAIN\*] ...

Task: What is the full query for using the TRAIN restriction?

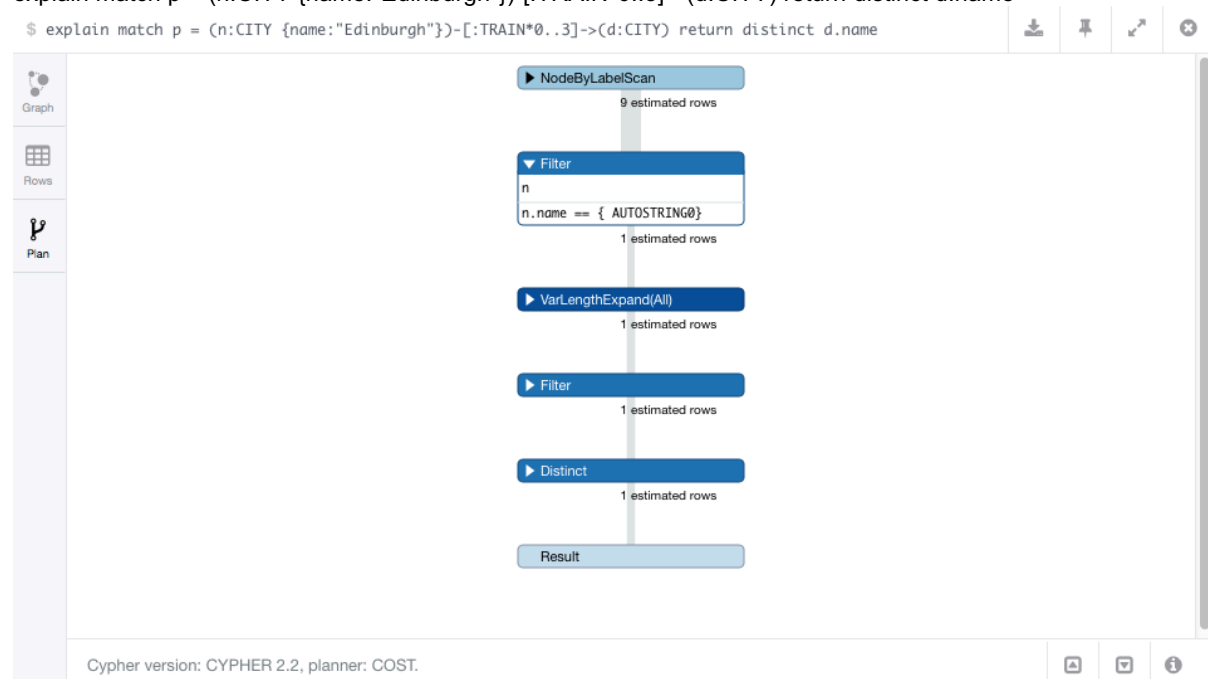
```
match p = (n:CITY {name:"Dundee"})-[:TRAIN*]->(d:CITY {name:"Cardiff"}) return
min(length(p)), max(length(p))
```

If we are willing to take no more than a 3 leg journey from Edinburgh, where can we get to?

```
match p = (n:CITY {name:"Edinburgh"})-[:TRAIN*0..3]->(d:CITY) return distinct d.name
```

To see optimise the execution of the query we must see the *query plan*:

```
explain match p = (n:CITY {name:"Edinburgh"})-[:TRAIN*0..3]->(d:CITY) return distinct d.name
```



Our query plan shows us that the *name* field is heavily used, so we should create an index on that field to help the database find names quickly:

```
create index on :CITY(name)
```

Task: What query can be used to find the destinations when you start from Edinburgh and take a journey with at least 2 legs but no more than 3?

```
match p = (n:CITY {name:"Edinburgh"})-[:TRAIN*2..3]->(d:CITY) return distinct d.name
```