Hands-On: Basic Queries in Neo4j

In this activity, we're going to go through a series of basic queries using Cypher with the focus on the data sets that we've already been using. Before starting, complete the following steps:

1. Start your Neo4j container and open it in https://localhost:7474/browser/.

2. If you started this activity just after finishing the *Hands-On: Importing Data into Neo4j* activity, make sure to "clean the slate" in Neo4j, as you probably have the terrorist data currently loaded, and for this activity, we will start with a smaller dataset:

First, run this code:

```
match (a)-[r]->() delete a,r;
match (a) delete a
```

3. Load the test.csv data

```
load CSV WITH HEADERS from "file:///datasets/test.csv" as line
merge (n:MyNode {Name: line.Source})
merge (m:MyNode {Name: line.Target})
merge (n) -[:TO {dist: toInteger(line.distance)}] -> (m)
```

Now we will start with some basic queries.

Query 1: Counting the number of nodes

```
match (n:MyNode)
return count(n)
```

The first line of code simply matches all of the nodes with the label MyNode, and then it returns a count of those nodes.

You can visually confirm your result by looking at the graph:

```
match (n:MyNode) return n
```

Query 2: Counting the number of edges

```
match (n:MyNode)-[r]->()
return count(r)
```

The first line of codes includes a declaration of the nodes associated with the edges, identifying them with the variable r, and then we're returning a count of those edges.

Query 3: Finding leaf nodes. Leaf nodes are defined as those nodes which have no outgoing edges.

```
match (n:MyNode) -[r:TO]->(m)
where not ((m)-->())
return m
```

We're matching all nodes associated with edges having the label TO and we want to place a constraint on those nodes, such that they have no outgoing edges. Then, we return all those nodes.

Query 4: Finding root nodes. Root nodes are defined as a node which has no incoming edges.

```
match (n) -[r:TO]->(m:MyNode)
where not ()-->(n)
return n
```

The first line of code is sort of a mirror image of the same segment in the first line of code from the previous query. We also place the constraint on the nodes that we want to return by specifying that they can have no incoming edges. Then, we return all of those nodes.

Query 5: Finding triangles. A triangle can be described as a three cycle, consisting of three nodes and three edges where the beginning and end node are the same.

```
match (a)-[:TO]->(b)-[:TO]->(c)-[:TO]->(a)
return distinct a, b, c
```

Here we're matching a node A which goes through an edge to node B, which goes through a second edge to a second node C and through a third edge back to the original node A. Then, we return all of those nodes.

Query 6: Finding 2nd neighbors of D. Second neighbors are nodes that are two nodes away from d.

```
match (a)-[:TO*..2]-(b)
where a.Name='D'
return distinct a, b
```

The first line of code matches all nodes that are two nodes away from a specific node. The second line of code specifies the actual node that we we want to consider by constraining its name to have the label *d*. Then, we return those nodes.

Additionally, we're using the command distinct because we want to make sure that we don't return any duplicate nodes. All of our nodes must be unique.

Some nodes appear to be only one node away from the node D but we can get to those nodes indirectly through another node, which means that they're not only a first neighbor but they're also a second neighbor.

For the next queries, we are going to switch again to the terrorist data. So again, first clean the state in Neo4j:

```
match (a)-[r]->() delete a,r;
match (a) delete a
```

And now load the terrorist data:

```
load CSV WITH HEADERS from "file:///datasets/terrorist_data_subset.csv" as row
merge (a:Actor {Name: row.Actor})
merge (t:Victim {Name: row.Victim, Aliases: row.Aliases, Type: row.ActorType})
merge (o:Organization {Names: row.Affiliations})
merge (a)-[:AFFILIATED_TO {Start: row.AffiliationStartDate, End: row.AffiliationEndDate}]->(o)
merge(v)-[:IS_FROM]-(a);
```

Query 7: Finding the types of a node.

```
match (n)
where n.Name = 'Afghanistan'
return labels(n)
```


Query 8. Finding the label of an edge

```
match (n {Name: 'Afghanistan'})-[r]-()
return distinct type(r)
```


Query 9. Finding all properties of a node

```
match (n:Actor)
return * limit 20
```


By clicking on *Text*, we can see that we returned the 20 first terrorists nodes with all of their properties.

Now we are going to work with a slightly different version of our road network. So we need to first reset our environment:

```
match (a)-[r]->() delete a,r;
match (a) delete a
```

And now we load the new version of the dataset:

```
load CSV WITH HEADERS from "file:///datasets/test2.csv" as line
merge (n:MyNode {Name: line.Source})
merge (m:MyNode {Name: line.Target})
merge (n) -[:TO {dist: line.distance}] ->(m)
```

Take a look at the data.

You will notice that in this new network, we have pairs of nodes with multiple edges connecting them, as well as nodes with edges connecting to themselves.

Query 10. Finding loops. If you remember from previous lectures, loops are defined as edges with connections to themselves.

```
match (n)-[r]->(n)
return n, r limit 10
```

In this query, the source and destination nodes are the same, so we return them along with the edges.

Query 11. Finding multigraphs. If you remember from previous lectures, a multigraph is any two nodes which have two or more edges between them.

```
match (n)-[r1]->(m), (n)-[r2]->(m)
where r1 <> r2
return n, r1, r2, m
```

In this case, we match two different node edge relationships. We apply a constraint that the edges must be different for the same pairs of nodes, and then we return those nodes and those edges.

Query 12. Finding the induced subgraph given a set of nodes. In our last query, we will return a subgraph.

```
match (x)-[r:TO]-(m)
where n.Name in ['A', 'B', 'C', 'D', 'E'] and m.Name in ['A', 'B', 'C', 'D', 'E']
return n, r, m
```

In this case, we provide a set of nodes, to return the network that consists only of those nodes and their associated edges.

Go to next item ✓ Completed