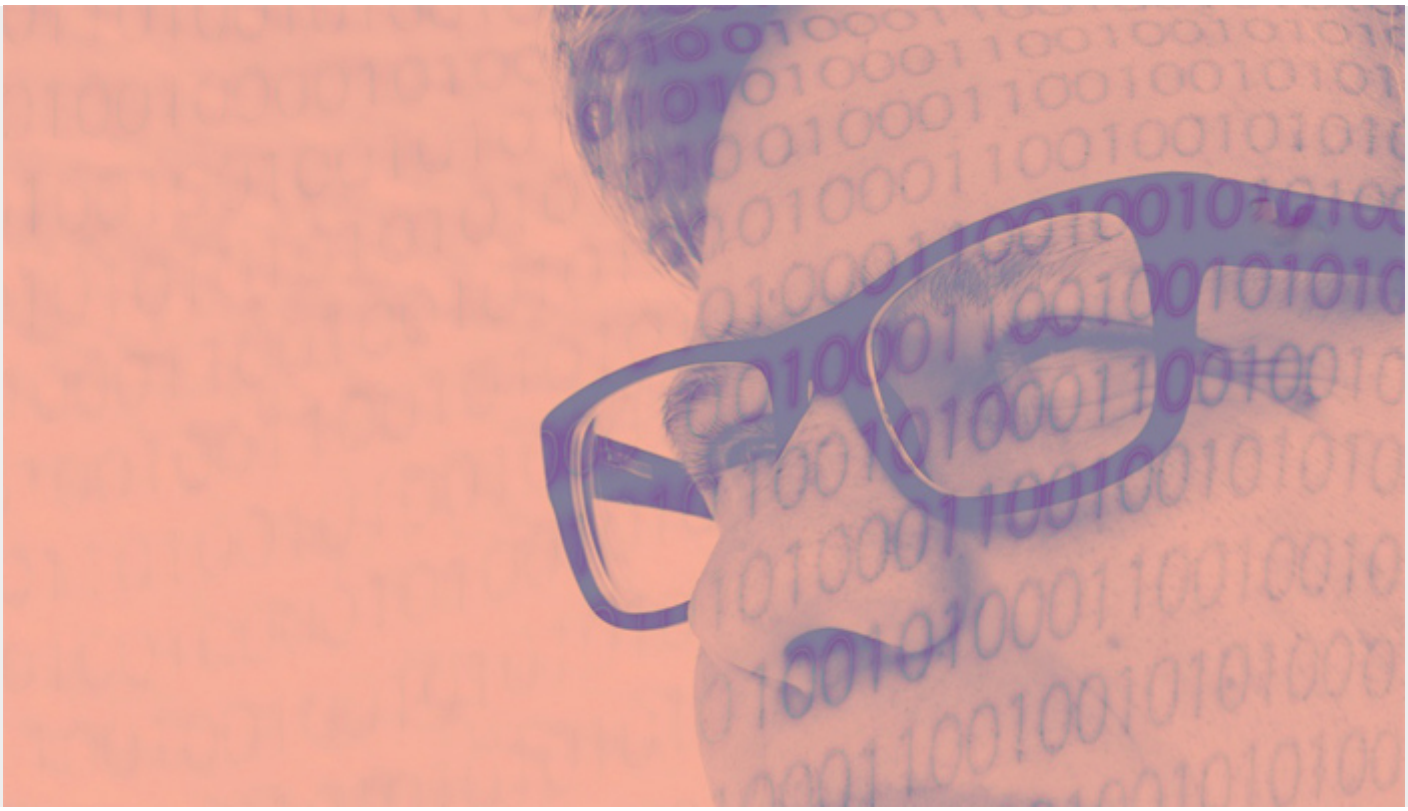
INTRODUCTION TO THE NEO4J GRAPH DATABASE



Neo4j is a graph database. A graph database, instead of having rows and columns has nodes edges and properties. It is more suitable for certain big data and analytics applications than row and column databases or free-form JSON document databases for many use cases.

A **graph database** is used to represent relationships. The most common example of that is the Facebook Friend relationship as well as the **Like** relationship. You can see some of that in the graphic below from Neo4j.

The circles are **nodes**. The lines, called **edges**, indicate relationships. And the any comments inside the circles are **properties** of that node.

graphic source: Neo4j

We write about Neo4j here because it has the largest market share. There are other players in this market. And according to Neo4J, Apache Spark 3.0 will add the Neo4j Cypher Query Language to allow and make easier "property graphs based on DataFrames to Spark." Spark already supports GraphX, which is an extension of the RDD to support Graphs. We will discuss that in another blog post.

In another post we will also discuss graph algorithms. The most famous of those is the Google Page Rank Index. Algorithms are the way to navigate the nodes and edges.

(This article is part of our Data Visualization Guide. Use the right-hand menu to navigate.)

Costs?

Is Neo4J free? That's rather complicated. The Community Edition is. So is the desktop version, suitable for learning. The Enterprise edition is not. That is consistent with other opensource products. When I asked Neo4J for a license to work with their product for an extended period of time they recommended that I use the desktop version. The Enterprise version has a 30-day trial period.

There are other alternatives in the market. The key would be to pick one that has enough users so that they do not go out of business. Which one should you use? You will have to do research to figure out that.

Install Neo4J

You can use the Desktop or tar version. Here I am using the tar version, on Mac. Just download it and then start up the shell as shown below. You will need a Java JDK, then.

```
export
JAVA_HOME='/Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home'
```

Start the server and set the initial password then open cypher-shell. The default URL is a rather strange looking **bolt://localhost:7687**.

```
cd neo4j bin folder

neo4j-admin set-initial-password xxxxxx

./cypher-shell   -a bolt://localhost:7687  -u neo4j  -p xxxxx
```

Create a Single Node

Thing of a node as a circle or an object, in the picture above. It can optionally have properties or a label or both.

Below is the simplest way to create a node. (Note that you have to end each command cypher command with a semicolon.) This node has nothing attached to it, meaning no properties or labels. It only has a variable, **n**, which you can use in other operations. What all of this means will become clear in subsequent steps.

```
create (n);
0 rows available after 13 ms, consumed after another 0 ms
Added 1 nodes
```

Cyper SQL is what Neo4j calls their command language. When you run a command it does not return any value unless you add the **return** keyword. In this example, as before, we create a node with nothing in it. Neo4j creates a variable, which we have called x, to represent that object which we can then return using the **return** statement, so that we can see it or use it in subsequent operations.

```
create(x) return x ;

+----+
```

```
| x  |
+----+
| () |
+----+
```

Create Multiple Nodes

You can create more than one node at a time:

```
create (o),(p);
0 rows available after 13 ms, consumed after another 0 ms
Added 2 nodes
```

Create a Node with a Label

A **label** is like the node name, or think of it as a type. Below is node with label **Student**. Labels have colons (:) in front of them. As before, s is just a variable. It's scope is the life of the shell. Close the shell and the variable goes away.

```
create (s:Student);
0 rows available after 16 ms, consumed after another 0 ms
Added 1 nodes, Added 1 labels
```

You can create nodes with more than one label:

```
create (s:Student:Biology) return s;
```

In addition to a label, a node can have **properties**, given as JSON:

```
CREATE (x:Employees { name: 'Walker', title: 'Tech Writer' })
CREATE (y:Employees { name: 'Stephen', title: 'Manager' })
return x,y;
```

Create a Relationship

The syntax for creating a relation is strange looking. Below, this example relationship is called **BOSS**. The **r** in front of it is just a variable. And the arrow shows the direction. You can use **->**, **<-**, or - to

indicate direction. Since it is complicated to remember direction you might want to use the - (dash), meaning both directions, in really complex models if direction does not matter. That's a design decision.

Below is how to create a relationship. We say in this example that Stephen is the **BOSS** of Walker. The **MATCH** statement is like the SQL select. It gives the selection criteria which shows which two elements should be connected to each other.

And here we use x and y as temporary variables. As you can see they let you use shorthand notation in subsequent statements, using the variable name instead of the :**label**.

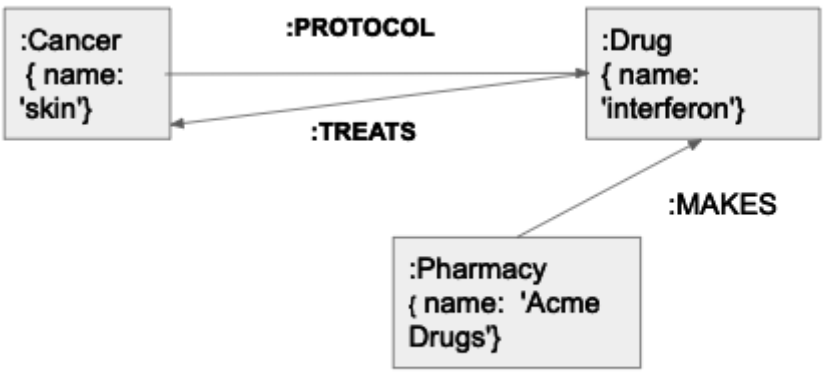
```
MATCH (x:Employees),(y:Employees)
WHERE y.name = "Stephen" AND x.name = "Walker"
CREATE (x)-->(y)
RETURN type(r);
```

Now, we carry the logic in the opposite direction. If Stephen is Walker's boss then Walker works for Stephen. So we can make Walker **EMPLOYEEOF** Stephen. We can use any descriptive name, like **EMPLOYEEOF**.

```
MATCH (x:Employees),(y:Employees)
WHERE y.name = "Stephen" AND x.name = "Walker"
CREATE (y)-->(x)
RETURN type(r);
```

Create Multiple Relationships

Here we give another example Here we have a model of drugs, what they are used for, and who makes them.



Create some data and then create the relationships. Pay attention to the output as if you make a mistake and the MATCH statement finds no data then the relationship will have no data too. In other words it should say **Create n nodes** or **Created n relationships**, where n > 0.

```
CREATE (x:Cancers { name: 'skin'});
CREATE (x:Drugs { name: 'interferon'});
CREATE (x:Pharmacy { name:  'Acme Drugs'});

MATCH (a:Cancers),(b:Drugs)
WHERE a.name = 'skin' AND b.name = 'interferon'
CREATE (a)-->(b), (b)-->(a);

MATCH (a:Pharmacy),(b:Drugs)
WHERE a.name = 'Acme Drugs' AND b.name = 'interferon'
CREATE (a)-->(b);
```

Here we make relationships in both directions, making two relationships at once. Notice that the **b** and **a** swap positions to show which belongs to which. We could also have pointed the arrow in the opposite direction.

```
MATCH (a:Cancers),(b:Drugs)
       WHERE a.name = 'skin' AND b.name = 'interferon'
       CREATE (a)-->(b), (b)-->(a);
```

Querying

Here we illustrate further how MATCH works. First we list all nodes with label **Pharmacy**.

```
MATCH (p:Pharmacy) return p;
+----------------------------------+
| p                                |
+----------------------------------+
| (:Pharmacy {name: "Acme Drugs"}) |
+----------------------------------+
```

Now we show all relationships of Pharmacies that make Drugs.

```
MATCH (p:Pharmacy)-->(b:Drugs) return p,b;
+-------------------------------------------------------------------+
| p                                | b                              |
+-------------------------------------------------------------------+
```

```
| (:Pharmacy {name: "Acme Drugs"}) | (:Drugs {name: "interferon"}) |
+----------------------------------------------------------------------+
```