

## Contents

Cypher Graph Query Language .....	2
VM Networking Configuration.....	3
Managing Databases .....	5
Introduction to Graphs .....	6
Graph Web Browser Commands .....	8
Creating Graphs .....	9
Relationships.....	11
Finding Graphs and Pattern Matching .....	12
Deleting Graphs .....	15

## Cypher Graph Query Language

The following is a helper lab for Cypher, the graph query language. The lab assumes you have a working Linux graph database server. As we have discussed, neo4j is a graph-based database that focuses on relationships in contrast to entities or tables. By prioritizing relationships first, it can perform better than traditional transactional databases when the number of joins between tables increases.

Neo4j documentation is well done and the basis for the installation as such is from the neo4j documentation. The credited/referenced tutorials lab are:

<https://neo4j.com/docs/getting-started/current/cypher-intro/>

<https://neo4j.com/docs/getting-started/current/appendix/graphdb-concepts/>

<https://neo4j.com/docs/cypher-manual/current/clauses/match/>

We also use our textbook for aspects of this lab.

Class Textbook: <https://github.com/PacktPublishing/Big-Data-Architects-Handbook>

## VM Networking Configuration

The first step is to make sure you enable host-only mode on your Ubuntu VM and allow all in promiscuous mode. This is necessary to access the neo4j web browser client.

Start your VM and ensure neo4j is running:

```
bobsmith@spark:~$ sudo systemctl status neo4j
[sudo] password for bobsmith:
● neo4j.service - Neo4j Graph Database
   Loaded: loaded (/lib/systemd/system/neo4j.s
   Active: active (running) since
   Main PID: 1137 (java)
```

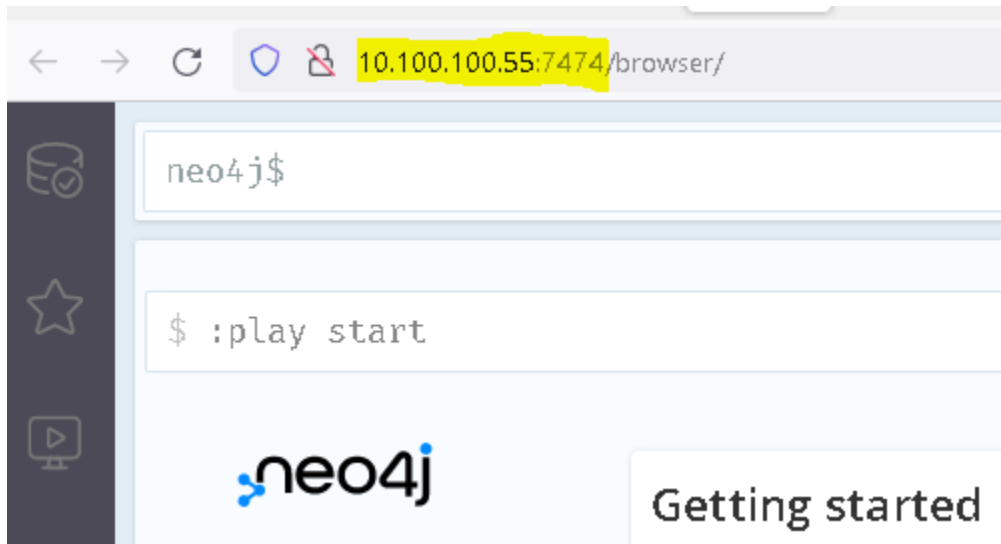
If necessary, troubleshoot with Netstat to ensure a relevant neo4j port is listening on the graph database server. You will connect to the listening port using a web browser.

```
bobsmith@spark:~$ netstat -tulpn | more
(Not all processes could be identified, non-c
will not be shown, you would have to be root
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           F
tcp        0      0 127.0.0.1:631            C
tcp        0      0 127.0.0.53:53            C
tcp        0      0 0.0.0.0:22                C
tcp6       0      0 :::7687                   :
tcp6       0      0 :::1:631                  :
tcp6       0      0 :::7474                   :
```

Find your IP address:

```
bobsmith@spark:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_
   link/loopback 00:00:00
   inet 127.0.0.1/8 scope
       valid_lft forever p
   inet6 ::1/128 scope ho
       valid_lft forever p
2: enp0s3: <BROADCAST,MULT
   link/ether 08:00:27:95
   inet 10.100.100.55/24
```

Next, connect to the neo4j client with a web browser using your specific IP address and port number.



Once connected, one of the first concepts we should learn is database management. Currently, the community edition does not allow multiple graph databases. Typically, we would create a new graph database with the CREATE DATABASE cypher syntax. We can see databases with the show command:

Show databases;

The screenshot shows the Neo4j client interface with the command 'system\$ show database;' entered in the terminal. The results are displayed in a table format. The table has columns for name, type, aliases, access, address, role, writer, and request. Two databases are listed: 'neo4j' and 'system'.

	name	type	aliases	access	address	role	writer	request
1	"neo4j"	"standard"	[]	"read-write"	"localhost:7687"	"primary"	true	"online"
2	"system"	"system"	[]	"read-write"	"localhost:7687"	"primary"	true	"online"

## Managing Databases

To use a new database use the “use: databasename” cypher.


```
neo4j$ :use neo4j
```



Use  
database

You have

Queries from this point and forward are using the database  
**neo4j** as the target.

Use the  : **db** to list all available databases.

To explore more detailed information for a given database enter:

```
SHOW DATABASE neo4j YIELD *;
```

## Introduction to Graphs

To begin to better understand the primary components of graph databases, ensure you complete the assigned textbook reading. As a starting point, let's review a graph with a few friend nodes and some relationships between the friends.

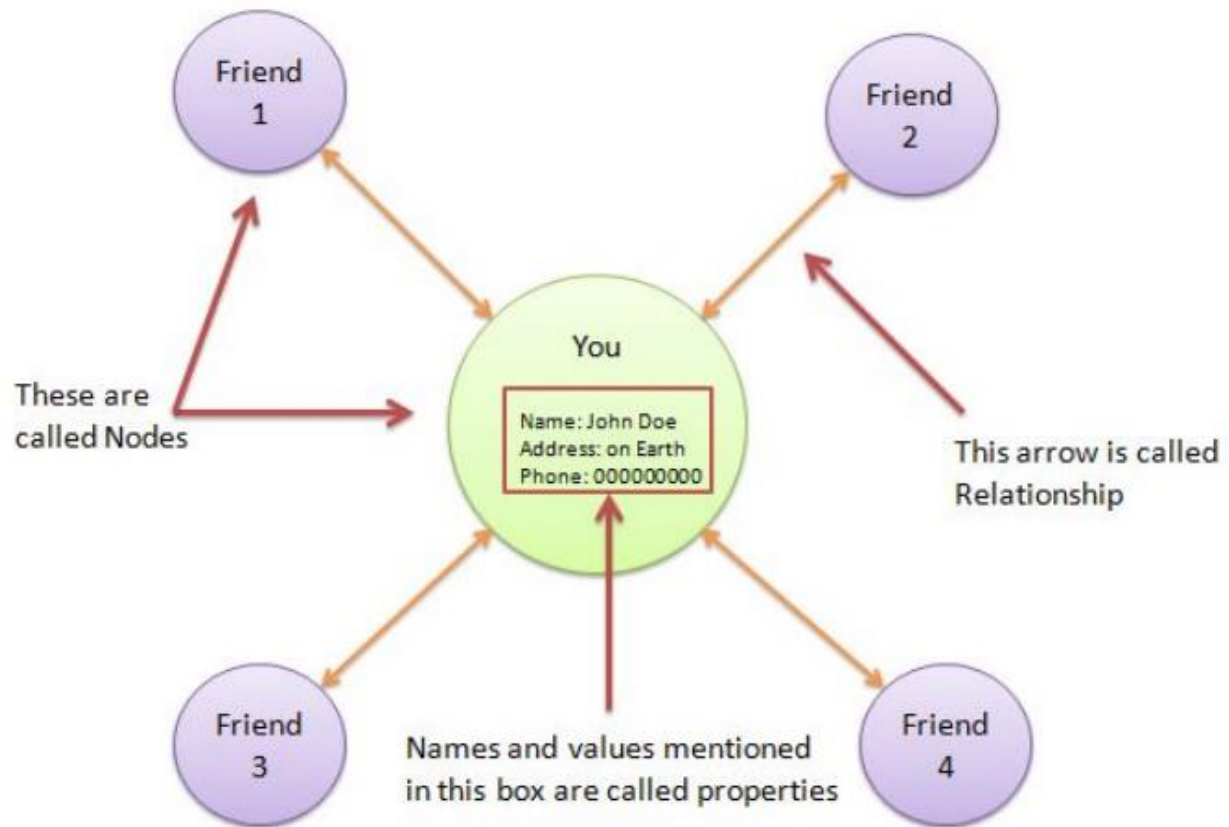


Figure 1. Graph Example from Diaz (2018)

Within Graph Theory, nodes can be a point or a vertex while edges or lines associate with relationships in neo4j.

Per our textbook, Diaz (2018) states, quote, "

There are three main components in a graph, namely:

**Node:** This is the primary object in representing any data in graph form. You may consider it like a main element that makes to other objects. It represents entities such as person, event, place, thing, and so on. If you take an example of a social networking graph, node is a representation of you and your friends connected to each other.

**Relationship:** It is the representation of how nodes are connected and what type of connection they are in. If we expand the earlier example, where you and your friends are connected to each other, relationship defines what type of connection it is—whether the connected one is a friend, colleague, business partner, and so on.

Properties: Every node and relationship may have some properties. For example, your nodes in the earlier example may contain some information related to you, such as your name, address, phone number, and so on. They are in the form of key-value pairs (Diaz, 2018, p. 115)."

We also have what are called "labels." Nodes can have zero labels, one label, or many labels. Labels tell us what kind of node the node is.

## Graph Web Browser Commands

To receive help during commands run the command:

`:help`

Within the web browser in neo4j, sometimes you want to add multiple lines without running the commands when you use the “Enter” key on your keyboard. Use the keyboard combination of Shift+Enter to start a new line and switch to multi-line mode.

If you switch to multi-line mode, the line numbers will appear and the “Enter” key on your keyboard will not run your query but will start a new line. You still need to run queries but to run queries in multi-line mode you use the “Ctrl+Enter” key combination.



## Creating Graphs

Nodes are typically distinguished by a round circle that represents a high-level object. Often, we use a noun, such as a person, place, or thing, to name a node. Parentheses are used to define nodes in Cypher. The basic syntax is:

```
CREATE (name_of_the_node_here);
```

To identify the name of nodes Cypher shell uses node labels. Labels are similar to tags or keywords to help find or filter various nodes in the graph.

To create a node with a variable “variable” and node label “Vehicle” use the syntax:

```
(variable:Vehicle)
```

Variables are similar to SQL, you can declare any variable and use it later in your cypher to reference the contents of the variable. A node can also be created without a variable, like:

```
(:Vehicle)
```

In addition to node labels, node properties exist. Node properties are similar to the data in a traditional database table row. They can be name-value pairs to provide more information about nodes and relationships between nodes. In contrast to parentheses used for nodes, properties are declared inside curly braces. For example:

```
(variable:Vehicle {name: 'Chevrolet'})
```

In this example, we declared a variable called “variable,” a node label called “Vehicle”, and a node property called “name”. Properties can have various data types like a field in a traditional database table. This can be a String like “Chevrolet” in the example or other types like Float, Integer, Date, and Point. Current cypher types are listed here: <https://neo4j.com/docs/cypher-manual/current/syntax/values/>

To begin testing your knowledge, create a node with multiple labels called “Person” and “Student” using your own name and graduate year as properties. Example code:

```
CREATE (:Person:Student {name: 'Amy Smith', YearGraduated: 2040});
```

Once you create your first node, you can use the MATCH syntax to find the new record.

```
MATCH (n:Person) RETURN (n);
```

```
neo4j$ MATCH (n:Person) RETURN n LIMIT 25
```

The image shows the Neo4j web interface. At the top, a Cypher query is entered: `neo4j$ MATCH (n:Person) RETURN n LIMIT 25`. Below the query bar, on the left, is a sidebar with icons for Graph, Table, Text, and Code. The main area displays a graph visualization of a single node, 'Amy Smith', which is highlighted in orange. This node is surrounded by a grey circular area divided into four quadrants, each containing an icon: a lock, a person, a graduation cap, and a network diagram. To the right of the graph, the 'Node properties' panel is visible. It shows two labels, 'Person' and 'Student', with 'Student' selected. Below the labels, the properties are listed: '<id>' with value '4', 'YearGraduated' with value '2040', and 'name' with value 'Amy Smith'.

To break down our actions, we first used the `CREATE` cypher syntax to create a new node. The `CREATE (:Person)` code creates a node with a label of “Person”. In this case, we created a node for Person and Student in the same cypher. The properties of the node are “name: Amy Smith” and the graduation year.

Labels associate with domains in graph theory, which help classify, organize, and group nodes into sets. All nodes can belong to the same set. For example, a Student node can associate with all students whereas a Staff node can associate with all university staff. Regardless, over time labels help us define the roles of people. As a subsequent example, consider the cypher:

```
CREATE (:Student:Athlete:Musician {name: 'Leslie Smith', sport: 'Soccer', totalgoals: 2});
```

In this example, Leslie Smith can belong to the sets associated with the node labels of athletes, musicians, and students. This allows us to organize, classify, and easily find all musicians at the university, for example.

## Relationships

Relationships exist between nodes. Cypher syntax uses the greater than and less than symbols or arrows to define relationships. Arrows include `- - >` and `< - -`. Relationships can exist in both directions used just a dash dash or `- -` with no arrows. Followed by the dash, relationships are defined by brackets `[ ]`. The basic syntax is:

```
CREATE (node_one)-[:TYPE_OF_RELATIONSHIP]->(node_two);
```

As an example, a person node with the label “Leslie” is teammates with “Amy” through the soccer relationship of “TEAMMATES\_WITH”.

```
(Leslie) - [:TEAMMATES_WITH] - > (Amy)
```

Relationships also have properties like nodes called relationship properties. To create a relationship between Leslie and Amy that associates them through their participation in band:

```
(Leslie) - [relationshipvariable:IN_BAND_WITH {instrument: 'Trumpet', since: 2030}] - > (Amy)
```

In this example, the relationship is in the brackets `[ ]` while the creation of a new relationship property is in the curly braces `{ }`. The full syntax would be:

```
CREATE  
(Leslie)-[relationshipvariable:IN_BAND_WITH {instrument: 'Trumpet', since: 2030}]->(Amy)  
RETURN relationshipvariable;
```

We can also create new nodes and relationships together. If Leslie and Amy do not exist, the following syntax creates both their person nodes along with their relationship.

```
CREATE  
(leslie:Person { name: 'Leslie'})-  
[relationshipvariable:IN_BAND_WITH {instrument: 'Trumpet', since: 2030}]  
->(amy:Person { name: 'Amy'})  
RETURN relationshipvariable;
```

## Finding Graphs and Pattern Matching

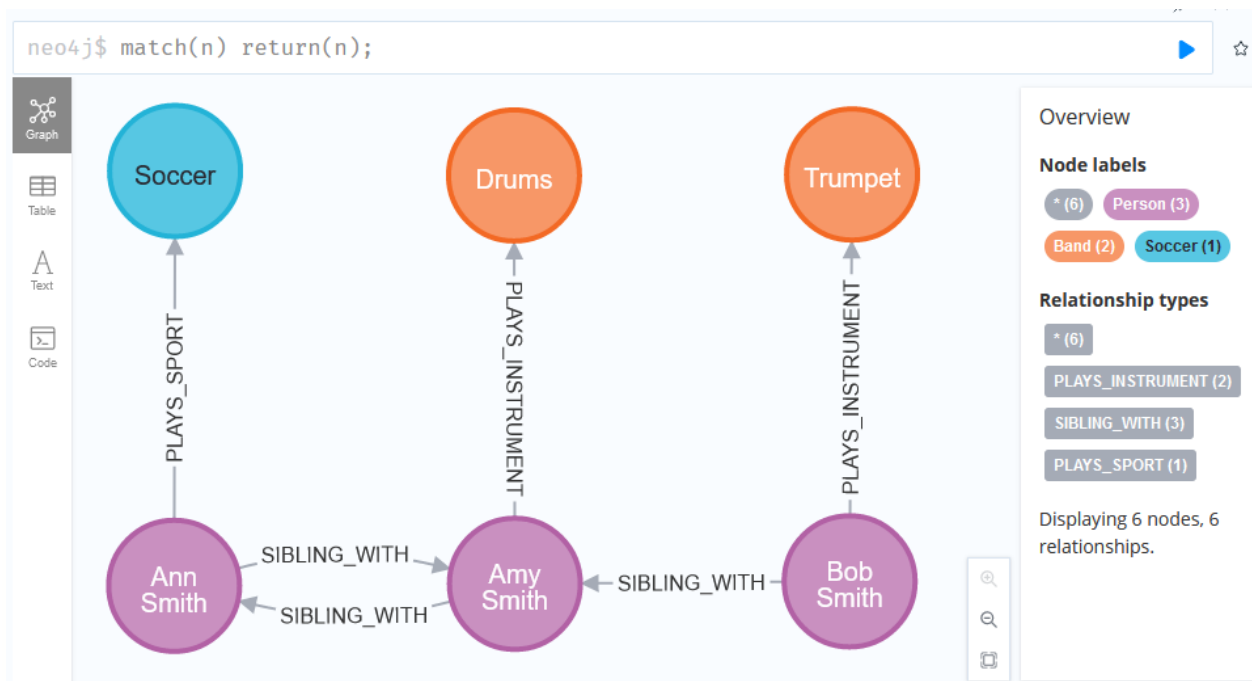
Because graphs rely heavily upon relationships, the MATCH cypher syntax is a critical aspect of many queries. MATCH allows pattern matching. It is typically paired with other syntax to find records, create records, delete records, etc. MATCH also accepts other functions and methods to find more specific patterns. For example, the WHERE clause similar to SQL allows finding more specific patterns when combined with the MATCH syntax.

To gain a better understanding, create the following graph for practice.

### CREATE

```
(amy:Person { name: 'Amy Smith', age: 30 } ),
(ann:Person { name: 'Ann Smith', age: 35 } ),
(bob:Person { name: 'Bob Smith', age: 32 } ),
(drums:Band { instrument: 'Drums', date: "2050-01-01" } ),
(trumpet:Band { instrument: 'Trumpet', date: "2040-02-02" } ),
(soccer:Soccer { sport: 'Soccer' } ),
(amy)-[:SIBLING_WITH]->(ann),
(ann)-[:SIBLING_WITH]->(amy),
(ann)-[:PLAYS_SPORT]->(soccer),
(bob)-[:SIBLING_WITH]->(amy),
(bob)-[:PLAYS_INSTRUMENT]->(trumpet),
(amy)-[:PLAYS_INSTRUMENT]->(drums)
```

View the Graph to get a better picture of the relationships.



Suppose we only want to find Band nodes. We create a variable called “variable” followed by the name of the node “Band”.

```
MATCH (variable:Band)
RETURN (variable)
```

If we want to find a specific instrument such as the trumpet, we first create a variable called “trumpet” followed by the node “Band”. Next, we can add a WHERE clause to find specific types of instrument using the “instrument” property within the Band node. We then return the variable:

```
MATCH (trumpet:Band)
WHERE trumpet.instrument = 'Trumpet'
RETURN (trumpet)
```

Oftentimes we need to modify an existing graph. This can be done using the MATCH syntax in cypher. As an example, Bob Smith does not currently play soccer. Only Amy does in the original graph. To add a relationship between Bob Smith and Soccer, we first need to find the two nodes.

```
MATCH (bob:Person), (soccer:Soccer)
WHERE bob.name = 'Bob Smith' AND soccer.sport = 'Soccer'
RETURN (bob), (soccer)
```

\*Note, the syntax may be underlined with squiggly lines and state something about a cartesian product between disconnected patterns. This is because the query is not optimized. Indexing is beyond the scope of this particular lab but normally we would want to use indexes in our matches to increase the pattern or search performance.

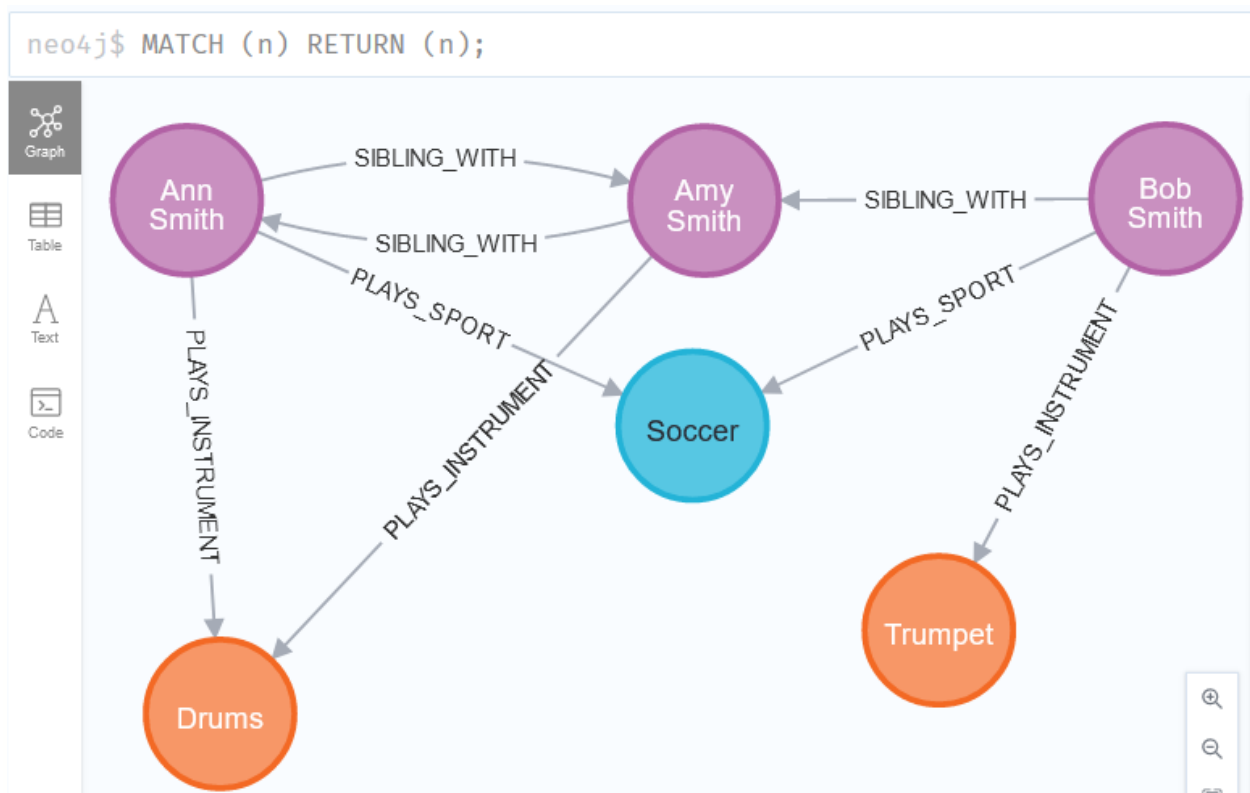
Once you have found the nodes for which you would like to create a new object like a relationship, you can simply add the action item below the MATCH and WHERE clauses. Thus, to associate Bob Smith with the sport Soccer:

```
MATCH (bob:Person), (soccer:Soccer)
WHERE bob.name = 'Bob Smith' AND soccer.sport = 'Soccer'
CREATE (bob)-[r:PLAYS_SPORT]->(soccer)
RETURN (r)
```

In this example, we first performed a pattern match to uniquely identify Bob Smith and the sport Soccer. Next, we created a relationship in the brackets [ r : PLAYS\_SPORT ]. Here the “r” is the variable so we can reuse the relationship. Notice we return the variable “r” to show the new relationship. The “PLAYS\_SPORT” is the relationship type.

Try to create a few other relationships and check your graph to see if they are being created the way you hope. If you create un-attached nodes, the match is not finding the proper nodes to create the relationships. In the example below, both Ann and Bob play soccer. In addition, both Amy and Ann play the drums.

```
MATCH (ann:Person), (drums:Band)
WHERE ann.name = 'Ann Smith' AND drums.instrument = 'Drums'
CREATE (ann)-[r:PLAYS_INSTRUMENT]->(drums)
RETURN (r)
```



## Deleting Graphs

To delete a node, we have to use the MATCH cypher to find the specific node to delete. Then, we have to detach the node from other relationships and delete it. The general syntax is:

```
MATCH (node:label {properties})  
DETACH DELETE node
```

If you are practicing and want to delete all nodes find all nodes first. To find all current nodes use the match syntax with (n):

```
MATCH (n) RETURN (n);
```

Using the prior match graph example, we can remove the relationship between Ann Smith and playing the instrument drums by first verifying the relationship:

```
MATCH (ann:Person)-[r:PLAYS_INSTRUMENT]->(drums:Band)  
WHERE ann.name = 'Ann Smith' AND drums.instrument = 'Drums'  
RETURN (r)
```

\*Note, similar to SQL, the WHERE clause should only match the records you want to delete. In this example, there should only be one matching relationship. If there are multiple matches in the return, perhaps the match is not accomplishing its goal. Thus, re-try other matches and troubleshoot until you get one match.

To delete nodes you have to detach them first and then delete them. Once the original match returns the correct relationship you want to delete, run it to see if it is removed.

```
MATCH (ann:Person)-[r:PLAYS_INSTRUMENT]->(drums:Band)  
WHERE ann.name = 'Ann Smith' AND drums.instrument = 'Drums'  
DETACH DELETE (r)  
RETURN (r)
```

If you want to delete all nodes you have to detach them first and then delete them. To delete all nodes you match all nodes.

**Be careful with the next syntax because it removes all of your nodes. Thus, only run this if you want to delete all of your prior work!**

```
MATCH (n) DETACH DELETE (n);
```