Neo4j is the world's leading open source Graph Database which is developed using Java technology. It is highly scalable and schema free (NoSQL).

## What is a Graph Database?

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. It is composed of two elements - nodes (vertices) and relationships (edges).

Graph database is a database used to model the data in the form of graph. In here, the nodes of a graph depict the entities while the relationships depict the association of these nodes.

### **Popular Graph Databases**

Neo4j is a popular Graph Database. Other Graph Databases are Oracle NoSQL Database, OrientDB, HypherGraphDB, GraphBase, InfiniteGraph, and AllegroGraph.

### **Why Graph Databases?**

Nowadays, most of the data exists in the form of the relationship between different objects and more often, the relationship between the data is more valuable than the data itself.

Relational databases store highly structured data which have several records storing the same type of data so they can be used to store structured data and, they do not store the relationships between the data.

Unlike other databases, graph databases store relationships and connections as first-class entities.

The data model for graph databases is simpler compared to other databases and, they can be used with OLTP systems. They provide features like transactional integrity and operational availability.

### **RDBMS Vs Graph Database**

Following is the table which compares Relational databases and Graph databases.

|  |  |  |
| --- | --- | --- |
| **Sr.No** | **RDBMS** | **Graph Database** |
| 1 | Tables | Graphs |
| 2 | Rows | Nodes |
| 3 | Columns and Data | Properties and its values |
| 4 | Constraints | Relationships |
| 5 | Joins | Traversal |

Neo4j is a NoSQL database. It is highly scalable and schema-free. It's world most popular graph database management system. Neo4j was developed by Neo technology and called an ACID-compliant transactional database with native graph storage and processing.

What is Neo4j

Neo4j is a NoSQL database. It is highly scalable and schema-free. It's world most popular graph database management system. Neo4j was developed by Neo technology and called an ACID-compliant transactional database with native graph storage and processing.

Neo4j is implemented in Java language and it can be accessed by other language using Cypher Query Language (CQL) through a transactional HTTP endpoint.

Neo4j is a way faster than traditional databases.

## Neo4j Working

* Neo4j stores and displays data in the form of graph. In Neo4j, data is represented by nodes and relationships between those nodes.
* Neo4j databases (as with any graph database) are a lot different to relational databases such as MS Access, SQL Server, MySQL, etc. Relational databases use tables, rows, and columns to store data. They also present data in a tabular fashion.
* Neo4j doesn't use tables, rows, or columns to store or present data.
* Neo4j is best for storing data that has many interconnecting relationships that's why graph databases like Neo4j has an advantage and much better at dealing with relational data than relational databases are.
* The graph model doesn't usually require a predefined schema. So there is no need to create the database structure before you load the data (like you do in a relational database). In Neo4j, the data is the structure. Neo4j is a "schema-optional" DBMS.
* In Neo4j, no need to set up primary key/foreign key constraints to predetermine which fields can have a relationship, and to which data. You just have to define the relationships between the nodes you need.

**Following is a list of most important features of Neo4j:**

**Highly scalable:** Neo4j is highly scalable. It provides a simple, powerful and flexible data model which can be changed according to applications and uses. It provides:

* Higher vertical scaling.
* Improved operational characteristics at scale.
* Higher concurrency.
* Simplified tuning.

**Schema-free:** Neo4j is schema-free like other NoSQL databases.

**High availability:** Neo4j provides high availability for large enterprise real-time applications with transactional guarantees.

**Real-time data analysis:** Neo4j provides results based on real-time data.

**Easy representation:** Neo4j provides a very easy way to represent connected and semi-structured data.

**Fast Execution:** Neo4j is fast because more connected data is very easy to retrieve and navigate.

**Easy retrieval:** Neo4j facilitates you not only represent but also easily retrieve **(traverse/navigate)** connected data faster other databases comparatively.

**Cypher Query language:** Neo4j provides CQL **(Cypher Query Language)** a declarative query language to represent the graph visually, using ASCII-art syntax. The commands of this language is very easy to learn and human readable.

**No Join:** Neo4j doesn't require complex Joins to retrieve connected/related data as it is very easy to retrieve its adjacent node or relationship details without Joins or Indexes because it is a graph database and all nodes are already connected.

Neo4j CQL Functions

A list of Neo4jCQL Function:

|  |  |  |
| --- | --- | --- |
| **Index** | **Function** | **Usage** |
| 1. | STRING | They are used to work with string literals. |
| 2. | Aggregation | They are used to perform some aggregation operations on CQL query results. |
| 3. | Relationship | They are used to get details of relationships such as startnode, endnode, etc. |

Neo4j CQL Data Types

The Neo4j CQL data types are similar to Java language data types. They are used to define properties of a node or a relationship.

A list of Neo4j CQL data types:

|  |  |  |
| --- | --- | --- |
| **Index** | **CQL Data Type** | **Usage** |
| 1. | Boolean | It is used to represent Boolean literals: True, False. |
| 2. | byte | It is used to represent 8-bit integers. |
| 3. | short | It is used to represent 16-bit integers. |
| 4. | int | It is used to represent 32-bit integers. |
| 5. | long | It is used to represent 64-bit integers. |
| 6. | float | Float is used to represent 32-bit floating-point numbers. |
| 7. | double | Double is used to represent 64-bit floating-point numbers. |
| 8. | char | Char is used to represent 16-bit characters. |
| 9. | String | String is used to represent strings. |

# **Neo4j CQL Operators**

Neo4j CQL Operators can be categorized in following types:

* **Mathematical Operators:**i.e. +, -, \*, /, %, ^
* **Comparison Operators:**i.e. +, <>, <, >, <=, >=
* **Boolean Operators:**i.e. AND, OR, XOR, NOT
* **String Operators:**i.e. +
* **List Operators:**i.e. +, IN, [X], [X?..Y]
* **Regular Expression:**i.e. =-
* **String matching:**i.e. STARTS WITH, ENDS WITH, CONSTRAINTS

Let's see the two most used Neo4j CQL Operators:

## Boolean Operators

Following is a list of Boolean operators which are used in Neo4j CQL WHERE clause to support multiple conditions:

|  |  |  |
| --- | --- | --- |
| **Index** | **Boolean operators** | **Description** |
| 1. | AND | It is a neo4j CQL keyword to support AND operation. It is like SQL AND operator. |
| 2. | OR | It is a Neo4j CQL keyword to support OR operation. It is like SQL AND operator. |
| 3. | NOT | It is a Neo4j CQL keyword to support NOT operation. It is like SQL AND operator. |
| 4. | XOR | It is a Neo4j CQL keyword to support XOR operation. It is like SQL AND operator. |

## Comparison Operators

A list of Neo4j CQL Comparison Operators used with WHERE clause:

|  |  |  |
| --- | --- | --- |
| **Index** | **Boolean operators** | **Description** |
| 1. | = | It is a Neo4j CQL "equal to" operator. |
| 2. | < > | It is a Neo4j CQL "not equal to" operator. |
| 3. | < | It is a Neo4j CQL "less than" operator. |
| 4. | > | It is a Neo4j CQL "greater than" operator. |
| 5. | <= | It is a Neo4j CQL "less than or equal to" operator. |
| 6. | > = | It is a Neo4j CQL"greater than or equal to" operator. |

# **GraphDB**

## What is Graph

A graph is a pictorial representation of objects which are connected by some pair of links. A graph contains two elements: Nodes (vertices) and relationships (edges).

## What is Graph database

A graph database is a database which is used to model the data in the form of graph. It stores any kind of data using:

* Nodes
* Relationships
* Properties

**Nodes:** Nodes are the records/data in graph databases. Data is stored as properties and properties are simple name/value pairs.

Nodes can be grouped together by applying a Label to each member. A node can have zero or more labels. Labels do not have any properties. Storing data in Neo4j is similar to add more records in other databases.

**Relationships:** It is used to connect nodes. It specifies how the nodes are related.

* Relationships always have direction.
* Relationships always have a type.
* Relationships form patterns of data.

**Properties:** Properties are named data values.

## Popular Graph Databases

Neo4j is the most popular Graph Database. Other Graph Databases are

* Oracle NoSQL Database
* OrientDB
* HypherGraphDB
* GraphBase
* InfiniteGraph
* AllegroGraph etc.

## Why GraphDB

Graph database is very useful now a day because in graph databases data exist in the form of the relationship between different objects. The relationship between the data is more valuable than the data itself.

Relational databases store highly structured data which have several records storing the same type of data so they can be used to store structured data and, they do not store the relationships between the data while graph databases store relationships and connections as first-class entities.

The data model for graph databases is simple compared to other databases and, they can be used with OLTP systems. They provide features like transactional integrity and operational availability.

# **Graph Database vs. RDBMS**

|  |  |  |
| --- | --- | --- |
| **Index** | **Graph Database** | **RDBMS** |
| 1. | In graph database, data is stored in graphs. | In RDBMS, data is stored in tables. |
| 2. | In graph database there are nodes. | In RDBMS, there are rows. |
| 3. | In graph database there are properties and their values. | In RDBMS, there are columns and data. |
| 4. | In graph database the connected nodes are defined by relationships. | In RDBMS, constraints are used instead of that. |
| 5. | In graph database traversal is used instead of join. | In RDBMS, join is used instead of traversal. |

# **Neo4j Data Model**

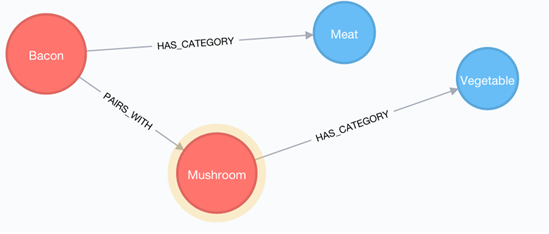
Neo4j Database follows the Property Graph Model for storing and managing its data. Neo4j is a graph database which contains the following features of Property Graph Model.

* The Graph model contains Nodes, Relationships and Properties which specifies data and its operation.
* Properties are key-value pairs.
* Nodes are represented using circle and Relationships are represented using arrow keys. Relationship specifies the relation between two nodes.
* There are two types of relationships between nodes according to their directions: Unidirectional and Bidirectional
* Each Relationship contains two nodes: "Start Node" or "From Node" and "To Node" or "End Node".
* Both Nodes and Relationships contain properties.

Relationships should be directional in Property Graph Data Mode. If you create a relationship without a direction, it will through an error message.

There are three main building block of a GraphDB Data model:

* Nodes
* Relationship
* Properties



# **What is Neo4j CQL**

CQL stands for **Cypher Query Language**. It is a query language for Neo4j just like SQL is a query language for Oracle or MySQL.

**Neo4j CQL Features**

* CQL is a query language for Neo4j Graph Database.
* Is a declarative pattern-matching language.
* The syntax of CQL is same like SQL syntax.
* Syntax of CQL is very simple and in human readable format.

**Similarity between Oracle SQL and Neo4j CQL**

* Oracle and Neo4j CQL both has simple commands to do database operations.
* Both support clauses like WHERE, ORDER BY, etc., to simplify complex queries.
* Oracle and Neo4j CQL supports some Relationship Functions and functions such as String, Aggregation.

# **Neo4j Create Nodes**

Node is a data or record in a graph database. In Neo4j, the CREATE statement is used to create a node. You can create the following things by using CREATE statement:

* Create a single node
* Create multiple nodes
* Create a node with a label
* Create a node with multiple labels
* Create a node with properties
* Returning the created node

## Create a Single Node

To create a single node in Neo4j, specify the name of the node along with CREATE statement.

**Syntax:**

1. **CREATE** (node\_name);

#### **Note: You can add or ignore semicolon (;). It is optional.**

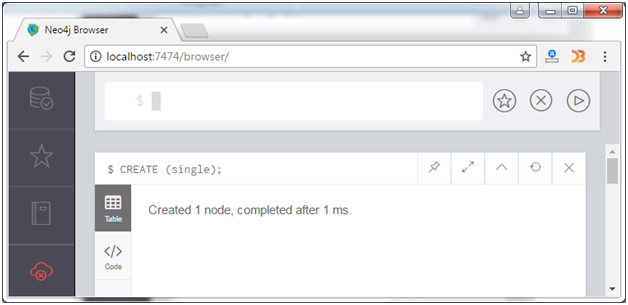
**Example1:**

Open the localhost on the browser: **http://localhost:7474/browser/** and use the following code:

1. **CREATE** (single);



Output:



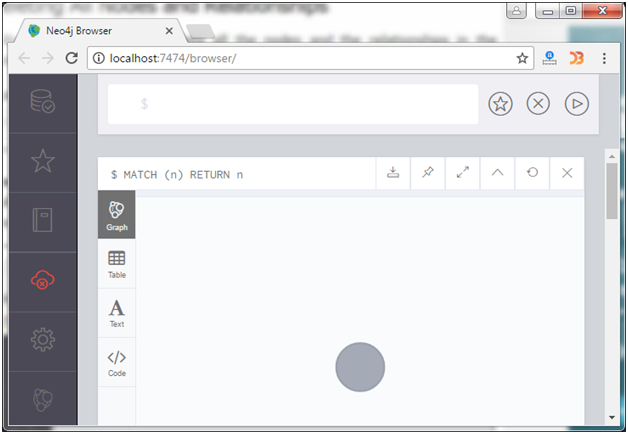
You can see that a node is created.

## Verification

Execute the following code to verify the creation of the node type:

1. MATCH (n) **RETURN** n

Output:



## Create Multiple Nodes

To create multiple nodes in Neo4j, use CREATE statement with the name of nodes separated by a comma.

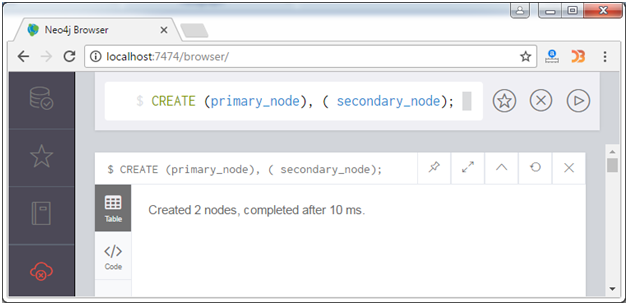
**Syntax:**

1. **CREATE** (node1),(node2), (node1),???..

**Example2:**

Let's create 2 nodes: primary\_node and secondary\_node.

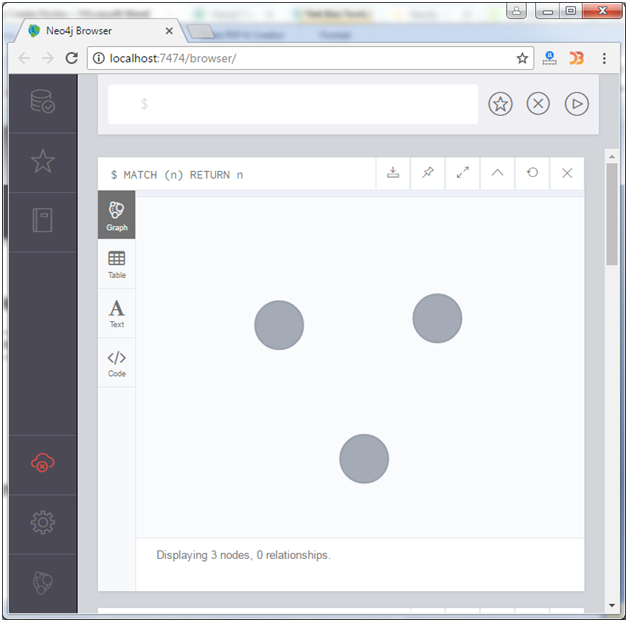
1. **CREATE** (primary\_node), ( secondary\_node);



## Verification

1. MATCH (n) **RETURN** n

Output:



#### **Note: It is displaying 3 nodes because we have created a node already in example1.**

## Create a node with a label

In Neo4j, a label is used to classify the nodes using labels. CREATE statement is used to create a label for a node in Neo4j.

**Syntax:**

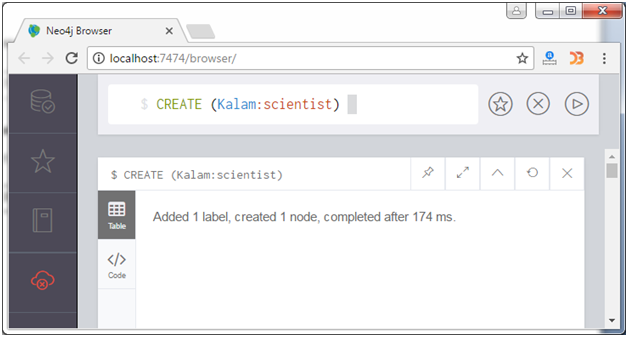
1. **CREATE** (node:label)

**Example3:**

Let's create a node "Kalam" with a label "scientist".

1. **CREATE** (Kalam:scientist)

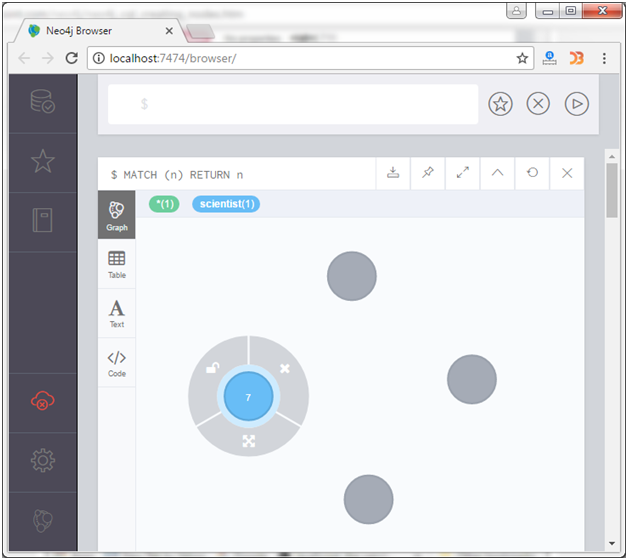
Output:



## Verification

1. MATCH (n) **RETURN** n

Output:



## Create a Node with Multiple Labels

To create multiple labels with a single node, you have to specify the labels for the node by separating them with a colon " : ".

**Syntax:**

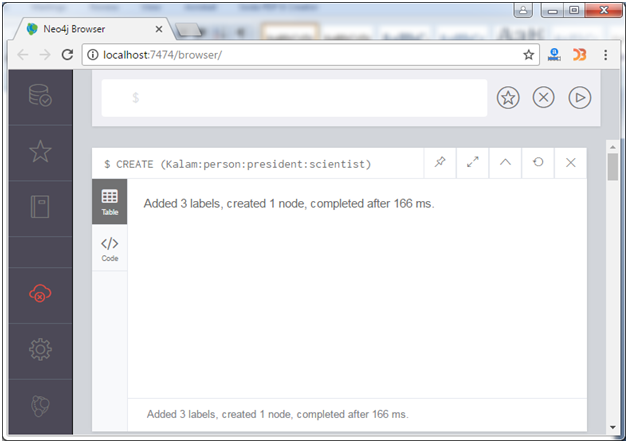
1. **CREATE** (node:label1:label2:. . . . labeln)

**Example:**

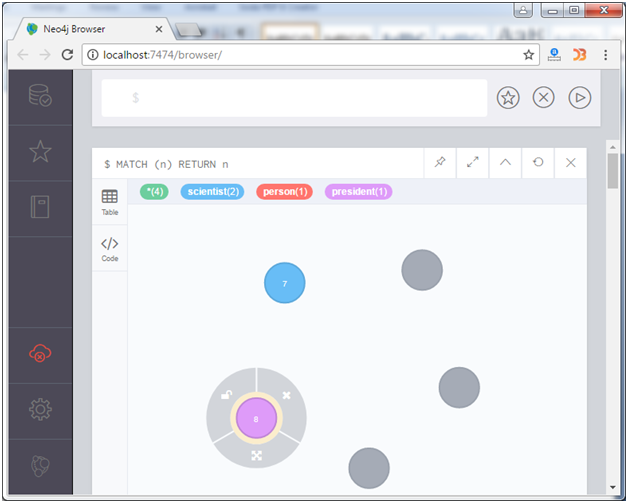
Create a node "Kalam" with label "person", "president", and "scientist".

1. **CREATE** (Kalam:person:president:scientist)

Output:



## Verification



## Create Node with Properties

In Neo4j, properties are the key-value pairs which are used by nodes to store data. CREATE statement is used to create node with properties, you just have to specify these properties separated by commas within the curly braces "{ }".

**Syntax:**

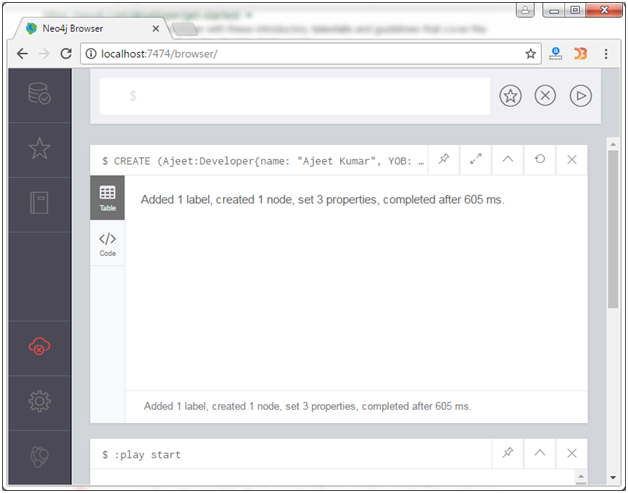
1. **CREATE** (node:label { key1: value, key2: value, . . . . . . . . .  })

**Example:**

Let's create a node "Ajeet", having the following properties:

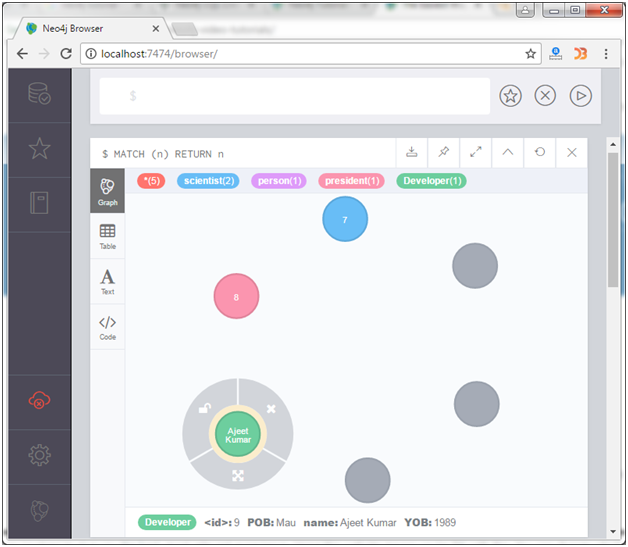
1. **CREATE** (Ajeet:Developer{**name**: "Ajeet Kumar", YOB: 1989, POB: "Mau"})

Output:



## Verification

1. MATCH (n) **RETURN** n



## Returning the created node

MATCH (n) RETURN n command is used to view the created nodes. This query returns all the existing nodes in the database.

But if you want to return the newly created node use the RETURN command with CREATE command:

**Syntax:**

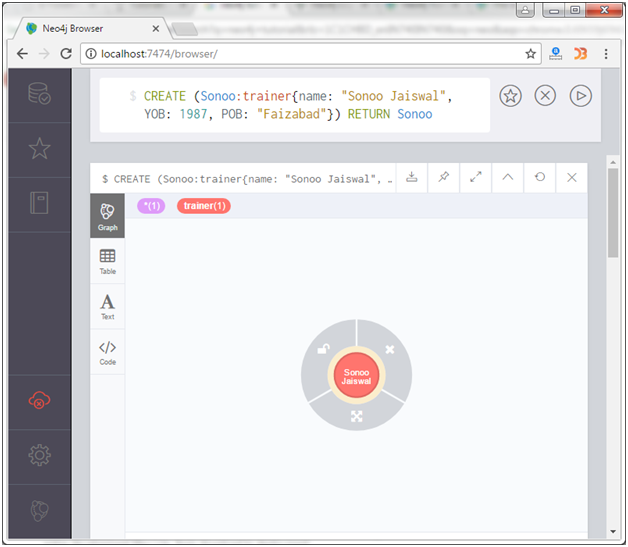
1. **CREATE** (Node:Label{properties. . . . }) **RETURN** Node

**Example:**

Create a node "Sonoo" with following properties and return that node.

1. **CREATE** (Sonoo:trainer{**name**: "Sonoo Jaiswal", YOB: 1987, POB: "Faizabad"}) **RETURN** Sonoo

Output:



# **Create Relationship**

CREATE statement is used to create relationship between nodes. These relationships define direction, type and form patterns of the data.

It defines mainly three things:

* Creating Relationships
* Creating Relationships between existing nodes
* Creating relationships with label and properties

## Creating Relationship

While creating a relationship, relationship should be specified within square braces "[ ]", depending on the direction of the relationship it is placed between hyphen " - " and arrow " ? " as shown in the following syntax.

**Syntax:**

6.6M

144

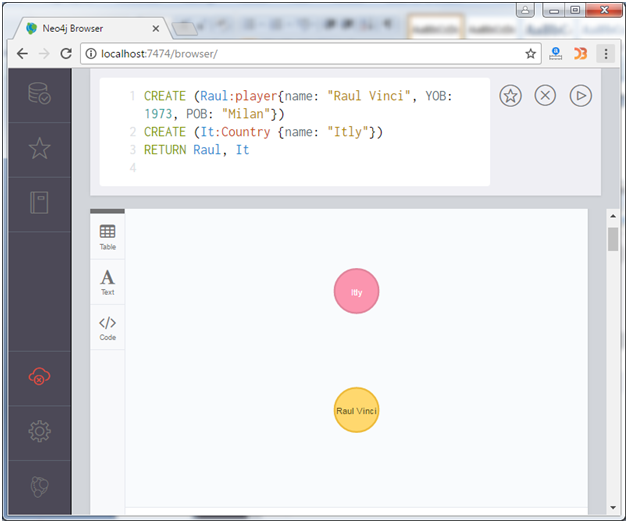
OOPs Concepts in Java

1. **CREATE** (node1)-[:RelationshipType]->(node2)

**Example**

Let's create two nodes "Raul" and "It" first and then specify the relationship between them.

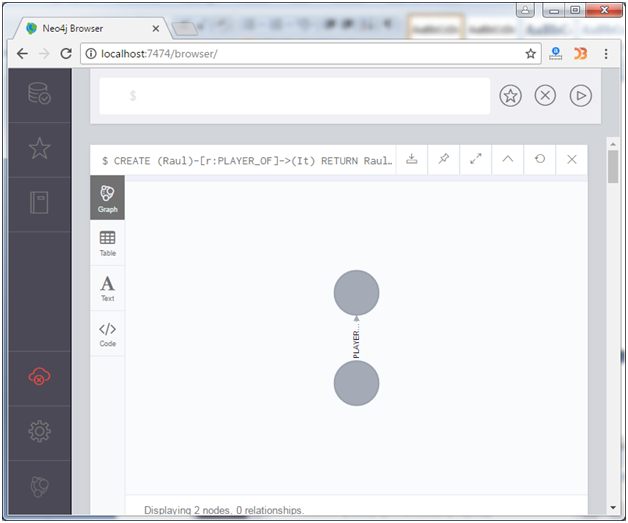
1. **CREATE** (Raul:player{**name**: "Raul Vinci", YOB: 1973, POB: "Milan"})
2. **CREATE** (It:Country {**name**: "Itly"})
3. **RETURN** Raul, It



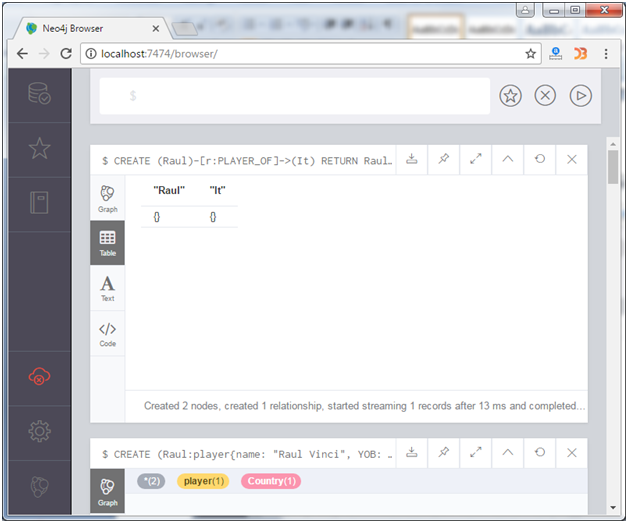
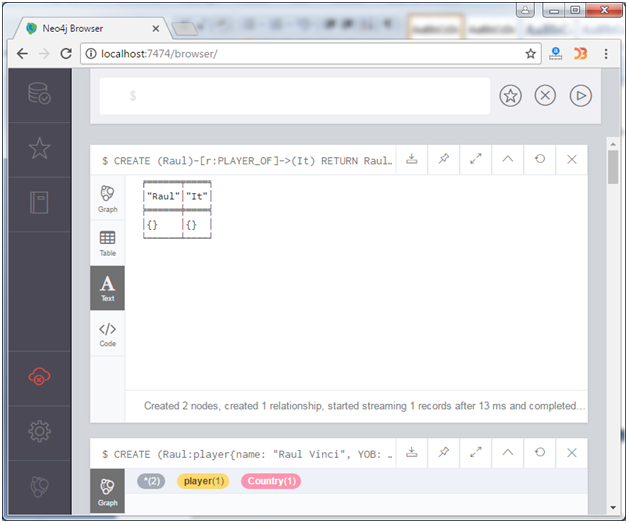
Now create a relationship "PLAYER\_OF between these two nodes as ?

1. **CREATE** (Raul)-[r:PLAYER\_OF]->(It)
2. **RETURN** Raul, It

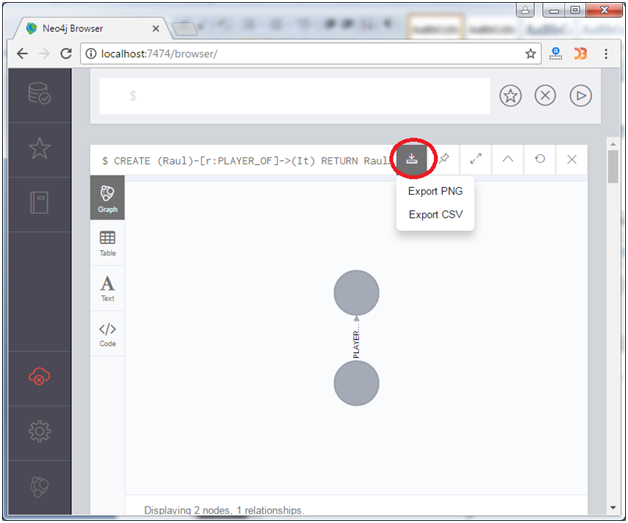
Output:



You can also see it in tabular or text form:

You can also download (export) the graph in which format you want to save it. Click on the download button, see the example:



## Create a Relationship between existing Nodes

MATCH statement is used to create relationship between the existing Nodes.

**Syntax:**

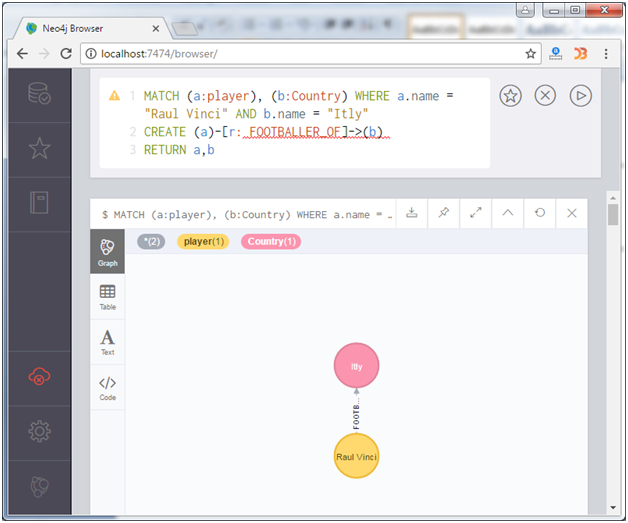
1. MATCH (a:LabeofNode1), (b:LabeofNode2)
2. **WHERE** a.**name** = "nameofnode1" AND b.**name** = " nameofnode2"
3. **CREATE** (a)-[: Relation]->(b)
4. **RETURN** a,b

**Example**

Create a relationship using MATCH statement.

1. MATCH (a:player), (b:Country) **WHERE** a.**name** = "Raul Vinci" AND b.**name** = "Itly"
2. **CREATE** (a)-[r: FOOTBALLER\_OF]->(b)
3. **RETURN** a,b

Output:



## Create a Relationship with Label and Properties

CREATE statement is used to create a relationship with label and properties.

**Syntax:**

1. **CREATE** (node1)-[label:Rel\_Type {key1:value1, key2:value2, . . . n}]-> (node2)

**Example:**

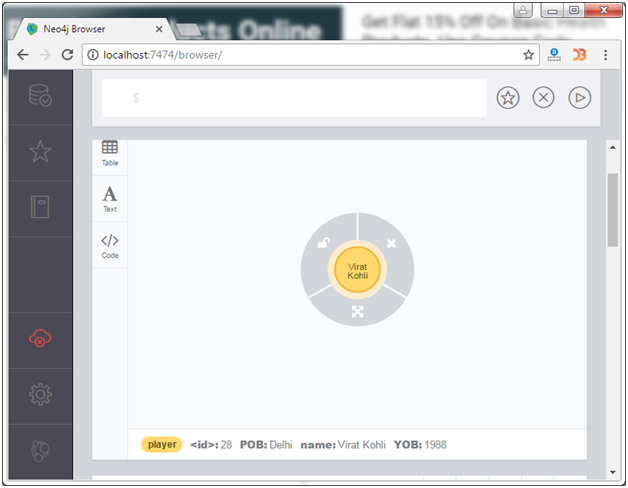
Let's take an example to create a relationship for a node with label and properties using the CREATE statement.

First create a node "Kohli", having multiple labels

1. **CREATE** (Kohli:person:player)

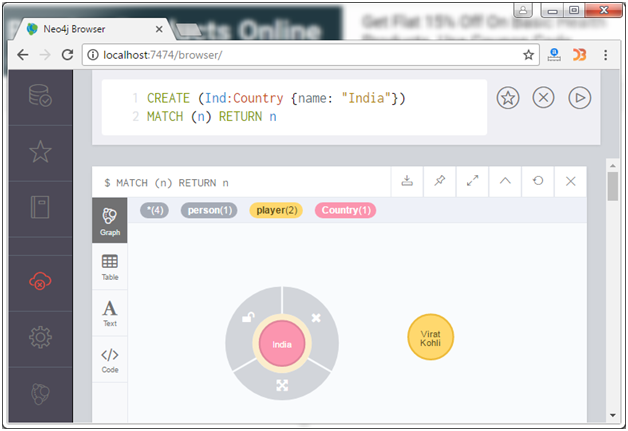
Then create some properties with the same node:

1. **CREATE** (Kohli:player{**name**: "Virat Kohli", YOB: 1988, POB: "Delhi"})
2. **RETURN** Kohli



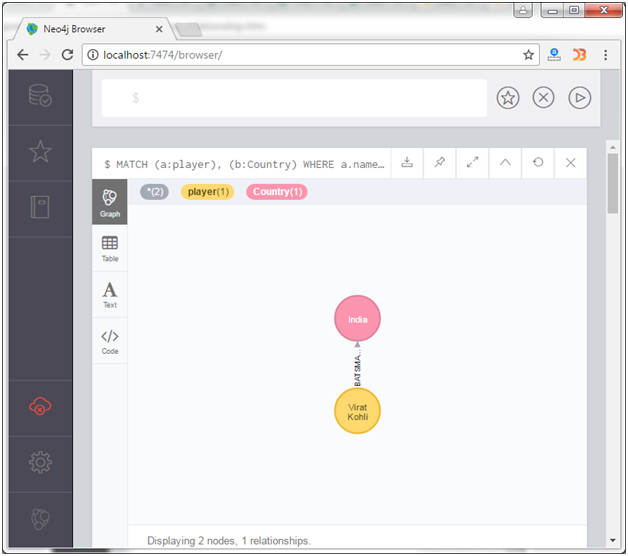
Create another node "Ind":

1. **CREATE** (Ind:Country {**name**: "India"})



Now create a relationship with label and properties:

1. MATCH (a:player), (b:Country) **WHERE** a.**name** = "Virat Kohli" AND b.**name** = "India"
2. **CREATE** (a)-[r:BATSMAN\_OF {Matches:5, Avg:90.75}]->(b)
3. **RETURN** a,b



## Creating a complete path

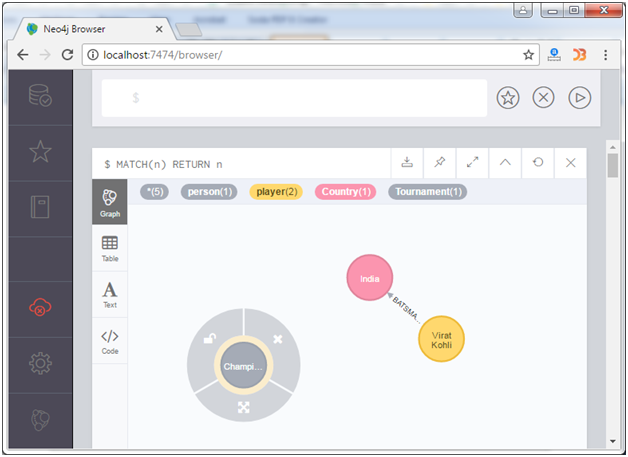
In Neo4j, CREATE statement is used to create a path. A path is formed using continuous relationship.

**Syntax:**

1. **CREATE** p = (Node1 {properties})-[:Relationship\_Type]->
2. (Node2 {properties})[:Relationship\_Type]->(Node3 {properties})
3. **RETURN** p

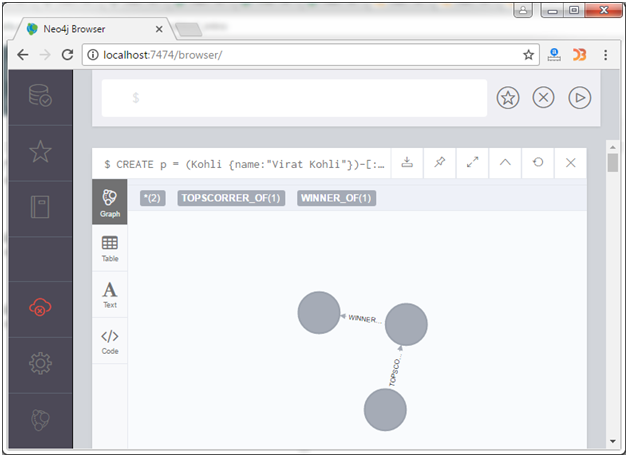
**Example:**

First create a node3 name "Champions\_Trophy" to do further operations.



Now execute the following code:

1. **CREATE** p = (Kohli {**name**:"Virat Kohli"})-[:TOPSCORRER\_OF]->
2. (Ind {**name**: "India"})-[: WINNER\_OF]->(Node3 {CT: "Champions\_Trophy"})
3. **RETURN** p



# **Select Data with MATCH**

MATCH statement is used to fetch the data which matches a given criteria. MATCH statement is also used to perform some operation and return the all nodes.

## Fetch a single Node

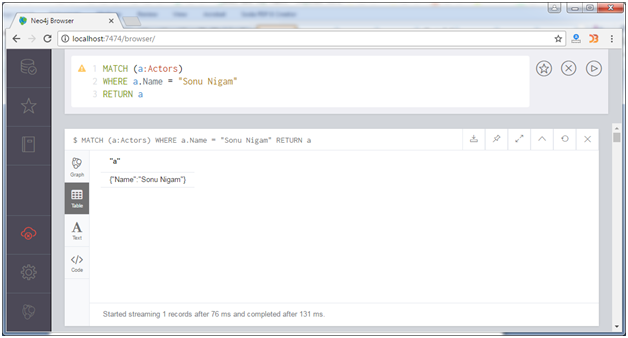
**Create a node:**

1. **CREATE** (a:Actors { **Name** : "Sonu Nigam" })

**Fetch single record:**

1. MATCH (a:Actors)
2. **WHERE** a.**Name** = "Sonu Nigam"
3. **RETURN** a

Output:



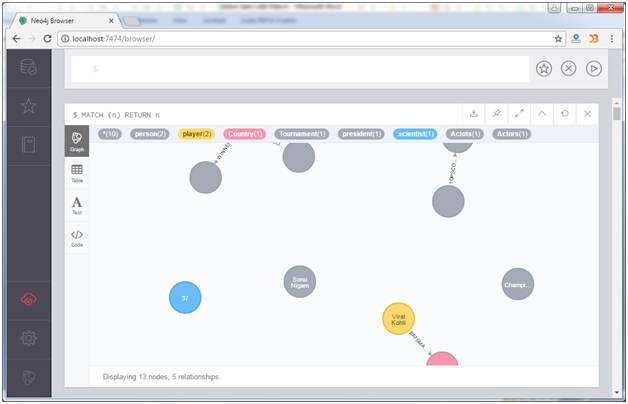
## Fetch all Nodes

If you want to retrieve all nodes from a database, just avoid the filters.

Use the following code to retrieve all nodes from a database:

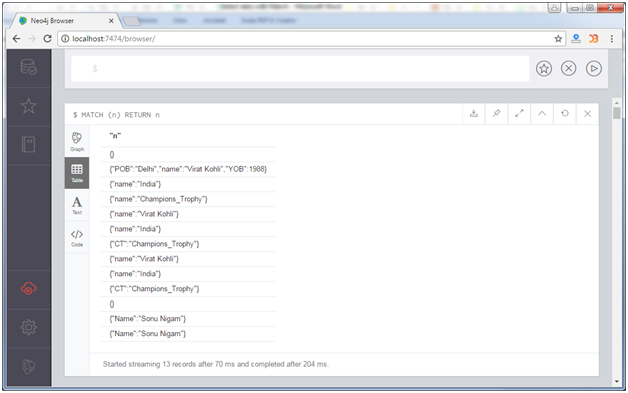
1. MATCH (n) **RETURN** n

Output:



**You can see it also in tabular form:**

**Tabular form:**

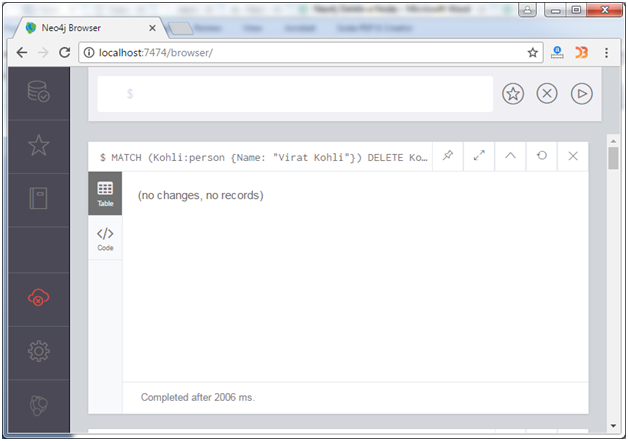


# **Neo4j Delete a Node**

In Neo4j, DELETE statement is always used with MATCH statement to delete whatever data is matched. The DELETE command is used in the same place we used the RETURN clause in our previous examples.

**Example**

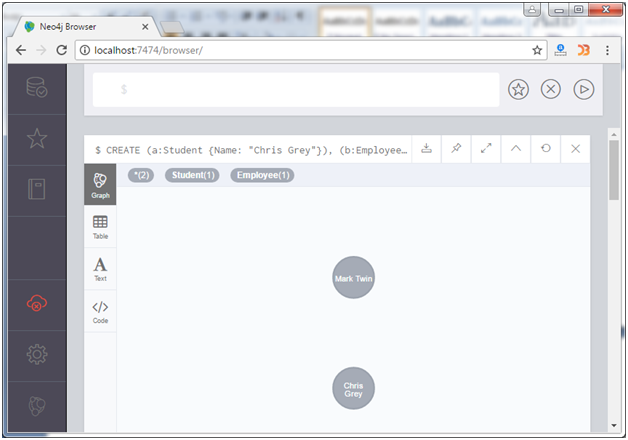
1. MATCH (Kohli:person {**Name**: "Virat Kohli"}) **DELETE** Kohli



## Delete Multiple Nodes

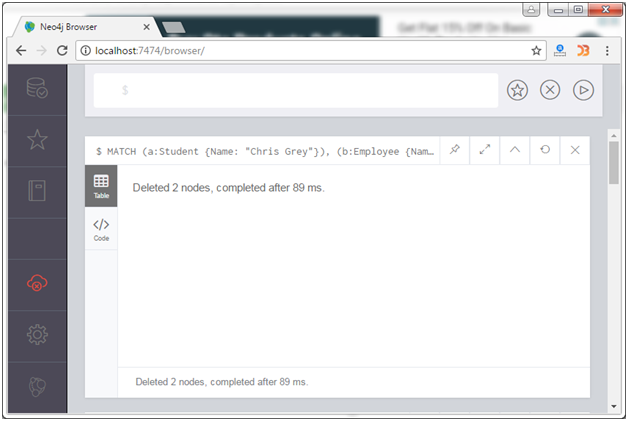
You can delete multiple nodes by using MATCH and DELETE commands in a single statement. You just have to put the different nodes separated by a column.

Suppose you have these two nodes:



Use the following command to delete the both nodes.

1. MATCH (a:Student {**Name**: "Chris Grey"}), (b:Employee {**Name**: "Mark Twin"})
2. **DELETE** a,b



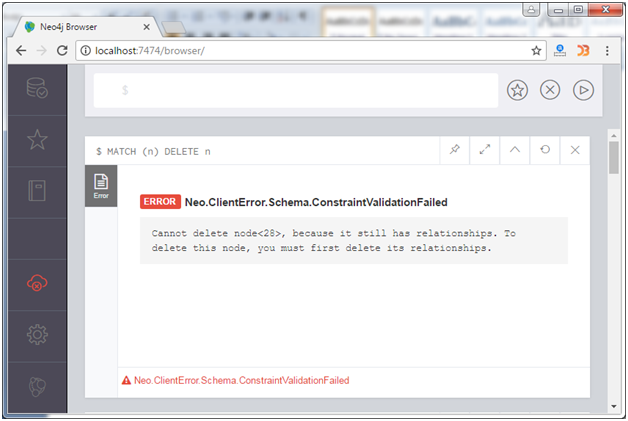
You can see the above displayed message that both nodes have been deleted.

## Delete All Nodes

To delete all nodes from the database, don?t use any filter criteria.

1. MATCH (n) **DELETE** n

#### **Note: The above statement cannot delete nodes if they have any relationships. In other words, you must delete any relationships before you delete the node itself. Otherwise you will get the following error message.**



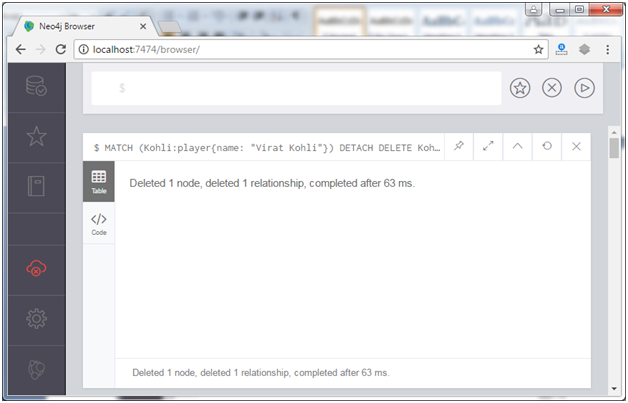
There is a method to delete a node and all relationships related to that node.

Use the DETACH DELETE statement:

**Example:**

1. MATCH (Kohli:player{**name**: "Virat Kohli"}) DETACH **DELETE** Kohli

Output:



This will delete the node "Kohli" where name is "Virat Kohli" with all its relationship.

# **Neo4j Delete a Relationship**

Deleting relationship is as simple as deleting nodes. Use the MATCH statement to match the relationships you want to delete.

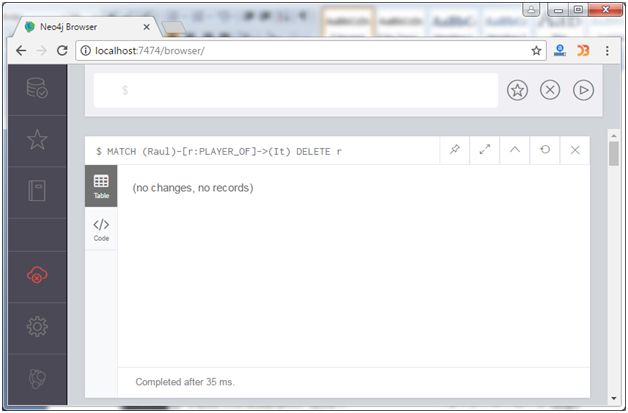
You can delete one or many relationships or all relationships by using one statement.

**Example:**

Delete the relationship named "PLAYER\_OF" from the database:

1. MATCH (Raul)-[r:PLAYER\_OF]->(It)
2. **DELETE** r

Output:



The above statement will match all Raul nodes that have a relationship type of PLAYER\_OF with an Itly node.

## Delete All Relationships related to a Node

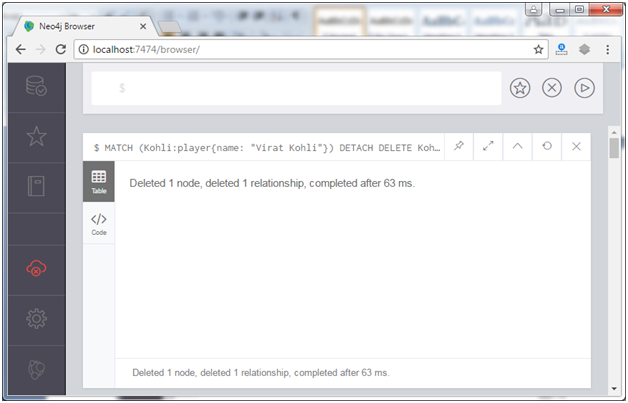
There is a method to delete a node and all relationships related to that node.

Use the DETACH DELETE statement:

**Example:**

1. MATCH (Kohli:player{**name**: "Virat Kohli"}) DETACH **DELETE** Kohli

Output:



This will delete the node "Kohli" where name is "Virat Kohli" with all its relationship.

## Delete Whole Database

To delete all database use the DETACH DELETE statement without using a filter.

1. DETACH **DELETE**;

## Create, Update, and Delete Operations

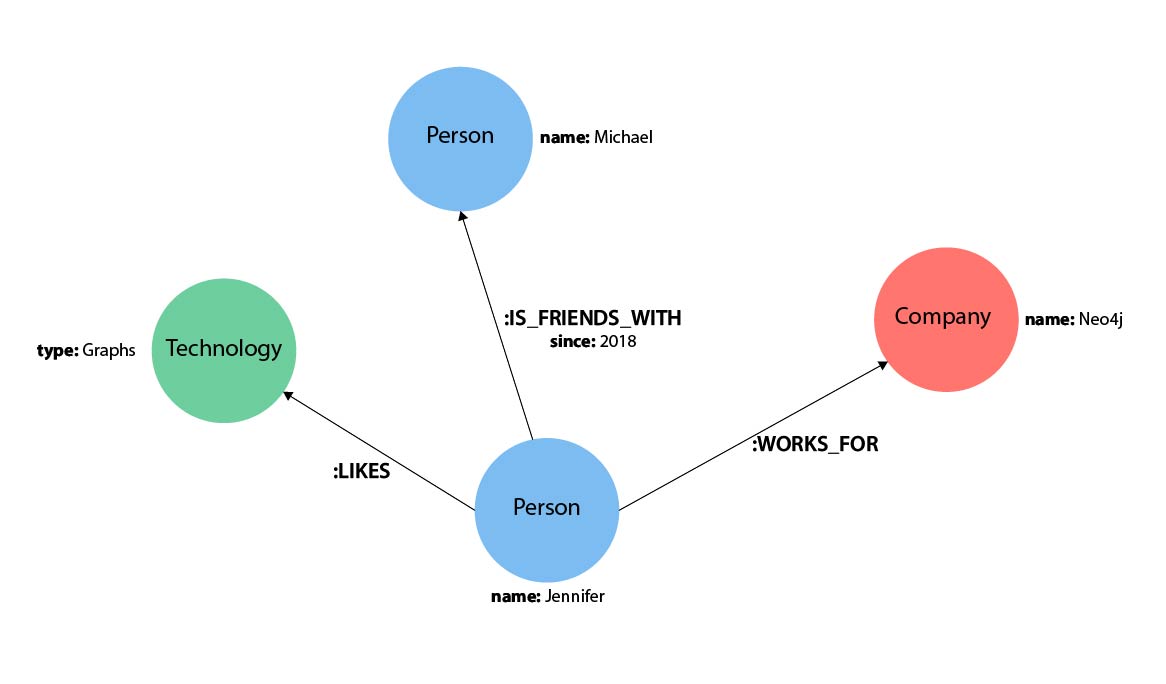
In the last guide, we learned how to represent nodes, relationships, labels, properties, and patterns in Cypher for read queries. This guide will add another level to your knowledge by introducing how to write create, update, and delete operations in Cypher.

While these are the standard CRUD operations, some things function a bit differently in a graph than in other types of databases. You will probably recognize some of the similarities and differences as we go along.

### Inserting Data with Cypher

Adding data in Cypher works very similarly to any other data access language’s insert statement. Instead of the INSERT keyword like in SQL, though, Cypher uses CREATE. You can use CREATE to insert nodes, relationships, and patterns into Neo4j.

Let us look at an example that we used in our last guide. To review, we had a Person node (Jennifer) who liked graphs, was friends with Michael, and worked at Neo4j.



What if we wanted to add another of Jennifer’s friends to the graph? We can add Jennifer’s friend Mark using the Cypher statement below.

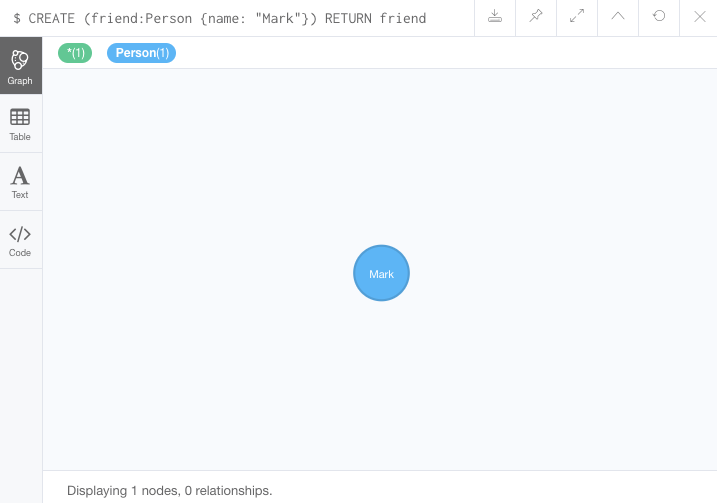
**Cypher**

Copy to ClipboardRun in Neo4j Browser

CREATE (friend:Person {name: 'Mark'})

RETURN friend

|  |
| --- |
| It is not required to include the RETURN clause in the Cypher statement above.  If you do not want to return any results, simply run this statement instead:  CREATE (friend:Person {name: 'Mark'}) |



Great! Now we added Mark to the database. However, Mark is all alone with no relationships because we just created his node and did not specify any connections. We know he is friends with Jennifer (same as Michael), so we can add a new IS\_FRIENDS\_WITH relationship between the existing Jennifer and Mark nodes. The Cypher to do that would look like this.

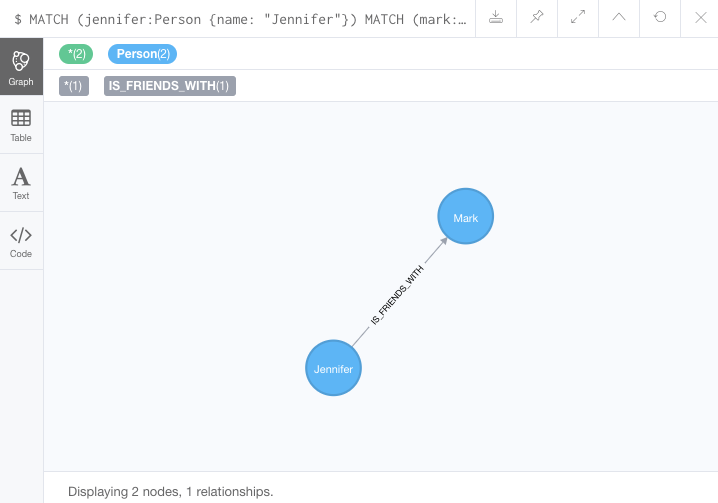
**Cypher**

Copy to ClipboardRun in Neo4j Browser

MATCH (jennifer:Person {name: 'Jennifer'})

MATCH (mark:Person {name: 'Mark'})

CREATE (jennifer)-[rel:IS\_FRIENDS\_WITH]->(mark)



Notice that we run two MATCH queries before we create a relationship between the nodes. Why is that? The reason we do a match for Jennifer’s node and a match for Mark’s node first is because the CREATE keyword does a blind insert and will create the entire pattern, regardless if it already exists in the database.

This means that running the Cypher statement below will insert duplicate Jennifer and Mark nodes. To avoid this, our previous query first found the existing nodes, and then created a new relationship between them.

**Cypher**

Copy to ClipboardRun in Neo4j Browser

*//this query will create duplicate nodes for Mark and Jennifer*

CREATE (j:Person {name: 'Jennifer'})-[rel:IS\_FRIENDS\_WITH]->(m:Person {name: 'Mark'})

## Updating Data with Cypher

Maybe you already have a node or relationship in the data, but you want to modify its properties. You can do this by matching the pattern you want to find and using the SET keyword to add, remove, or update properties.

Using our example thus far, we could update Jennifer’s node to add her birthday. The next Cypher statement shows how to do this. First, we need to find our existing node for Jennifer. Next, we use SET to create the new property (with syntax variable.property) and set its value. Finally, we can return Jennifer’s node to ensure that the information was updated correctly.

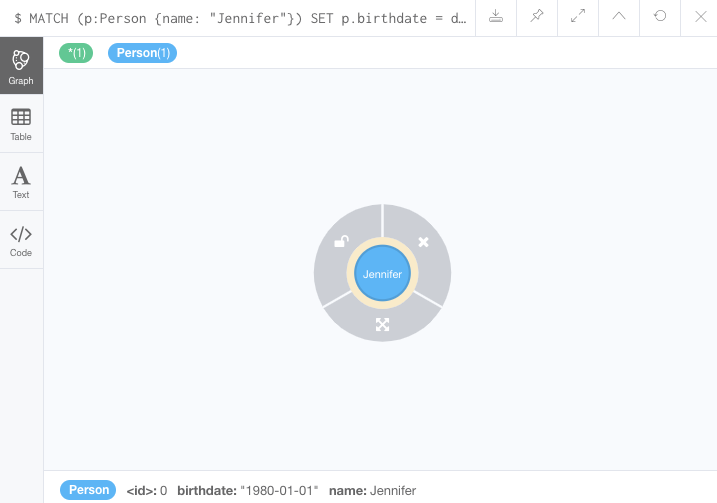
**Cypher**

Copy to ClipboardRun in Neo4j Browser

MATCH (p:Person {name: 'Jennifer'})

SET p.birthdate = date('1980-01-01')

RETURN p



If we now wanted to change her birthday, we could use the same query above to find Jennifer’s node again and put a different date in the SET clause.

We could also update Jennifer’s WORKS\_FOR relationship with her company to include the year that she started working there. To do this, you can use similar syntax as above for updating nodes.

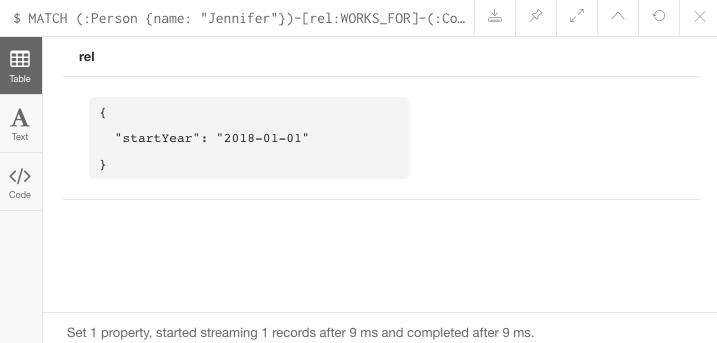
**Cypher**

Copy to ClipboardRun in Neo4j Browser

MATCH (:Person {name: 'Jennifer'})-[rel:WORKS\_FOR]-(:Company {name: 'Neo4j'})

SET rel.startYear = date({year: 2018})

RETURN rel



|  |
| --- |
| If we wanted to return a graph view on the above query, we could add variables to the  nodes for p:Person and c:Company and  write the return line as RETURN p, rel, c. |

## Deleting Data with Cypher

Another operation for us to cover is how to delete data in Cypher. For this operation, Cypher uses the DELETE keyword for deleting nodes and relationships. It is very similar to deleting data in other languages like SQL, with one exception.

Because Neo4j is ACID-compliant, you cannot delete a node if it still has relationships. If you could do that, then you might end up with a relationship pointing to nothing and an incomplete graph. We will walk through how to delete a disconnected node, a relationship, as well as a node that still has relationships.

### Delete a Relationship

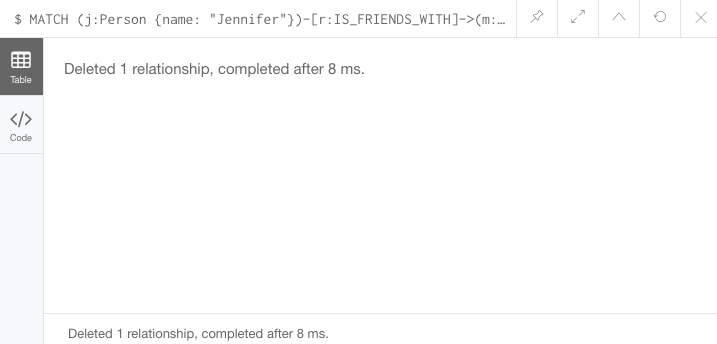
To delete a relationship, you need to find the start and end nodes for the relationship you want to delete and then use the DELETE keyword, as shown in the statement below. Let us go ahead and delete the IS\_FRIENDS\_WITH relationship between Jennifer and Mark for now. We will add this relationship back in a later exercise.

**Cypher**

Copy to ClipboardRun in Neo4j Browser

MATCH (j:Person {name: 'Jennifer'})-[r:IS\_FRIENDS\_WITH]->(m:Person {name: 'Mark'})

DELETE r



### Delete a Node

To delete a node that does not have any relationships, you need to find the node you want to delete and then use the DELETE keyword, just as we did for the relationship above. We can delete Mark’s node for now and add him back in a later exercise.

**Cypher**

Copy to ClipboardRun in Neo4j Browser

MATCH (m:Person {name: 'Mark'})

DELETE m



### **Delete a Node and Relationship**

Instead of running the last two queries to delete the IS\_FRIENDS\_WITH relationship and the Person node for Mark, we can actually run a single statement to delete the node and relationship at the same time. As we mentioned above, Neo4j is ACID-compliant so it doesn’t allow us to delete a node if it still has relationships. Using the DETACH DELETE syntax tells Cypher to delete any relationships the node has, as well as remove the node itself.

The statement would look like the code below. First, we find Mark’s node in the database. Then, the DETACH DELETE line removes any existing relationships Mark has before also deleting his node.

**Cypher**

Copy to ClipboardRun in Neo4j Browser

MATCH (m:Person {name: 'Mark'})

DETACH DELETE m

### **Delete Properties**

You can also remove properties, but instead of using the DELETE keyword, we can use a couple of other approaches. The first option is to use REMOVE on the property. This tells Neo4j that you want to remove the property from the node entirely and no longer store it.

The second option is to use the SET keyword from earlier to set the property value to null. Unlike other database models, Neo4j does not store null values. Instead, it only stores properties and values that are meaningful to your data. This means that you can have different types and amounts of properties on various nodes and relationships in your graph.

To show you both options, let us look at the code for each.

**Cypher**

Copy to ClipboardRun in Neo4j Browser

*//delete property using REMOVE keyword*

MATCH (n:Person {name: 'Jennifer'})

REMOVE n.birthdate

*//delete property with SET to null value*

MATCH (n:Person {name: 'Jennifer'})

SET n.birthdate = null



# **Reference:**

# [**https://neo4j.com/docs/cypher-manual/current/clauses/set/#set-update-a-property**](https://neo4j.com/docs/cypher-manual/current/clauses/set/#set-update-a-property)

# [**https://neo4j.com/developer/cypher/updating/**](https://neo4j.com/developer/cypher/updating/)

# [**https://neo4j.com/videos/getting-started-with-neo4j-desktop-1-2-7-on-windows/**](https://neo4j.com/videos/getting-started-with-neo4j-desktop-1-2-7-on-windows/)

# 