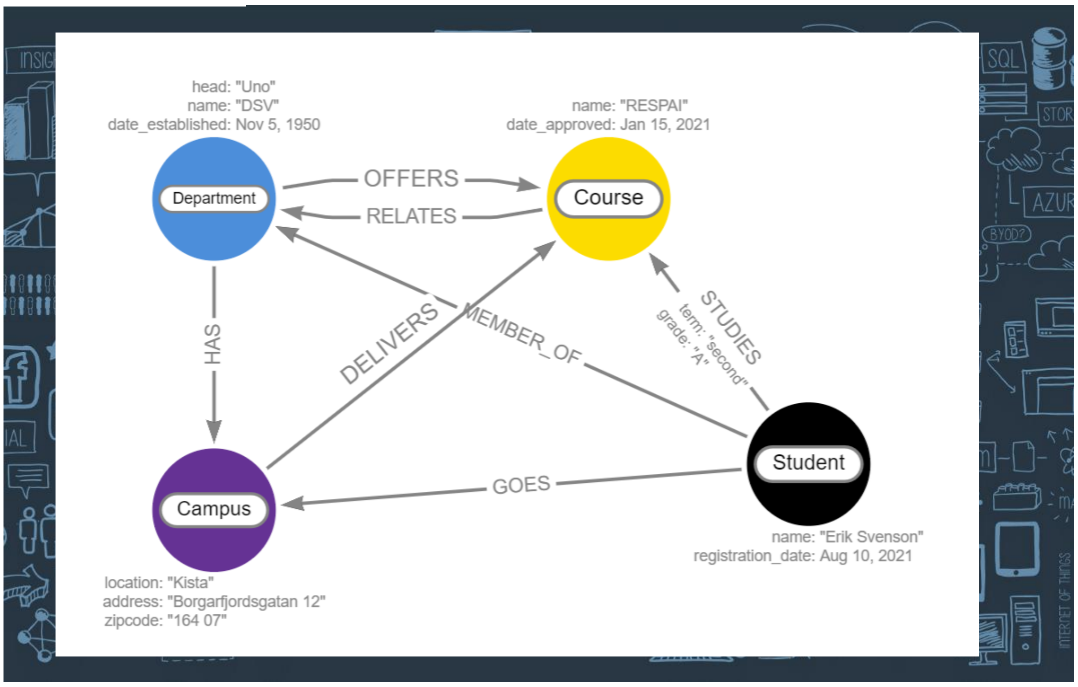


Preparing for Assignment 1, part 1 – Getting started with the graph query language in Neo4j (cypher)

RESPAI

Scalable and Responsible AI in Organizations | VT-2024
Compiled by Workneh Yilma Ayele

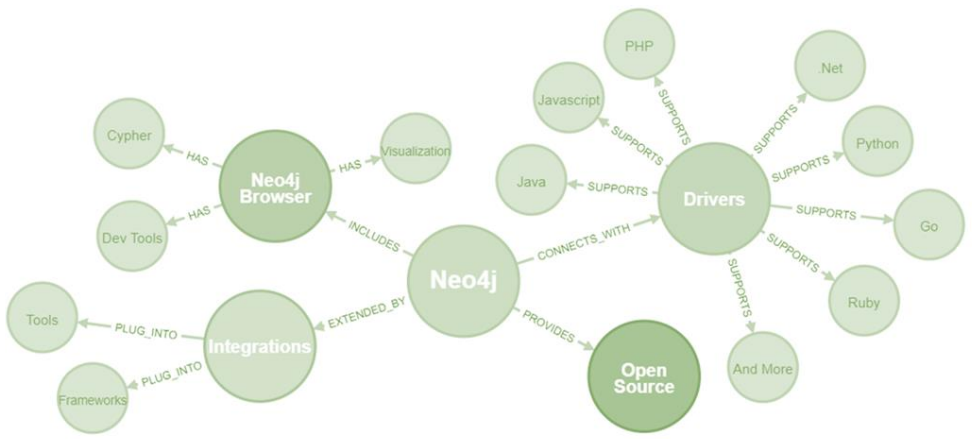


Table of Contents

Objectives of this tutorial

- Familiarize yourself with graph database
- Familiarize yourself with graph database basic concepts
- Learn about Neo4j Cypher to create and manage graph database
- Learn about Neo4j Bloom for more enhanced visualization of graph data analytics

1. Introduction – What is a Graph Database?

A collection of vertices (nodes) and edges (relationships) that connects vertices, where nodes represent conceptual or real world entities and their connections as relationships (Robinson et al. 2015). Graphs have become necessary these days, and their importance is attributed to our quest for information and the capabilities of graph technologies to address our needs seamlessly. Besides, our inquiries are not easily answered without implementing complex joints from legacy database systems. Analysis of relationships of nodes answers our quests for valuable information. Thus, relationships are more crucial in graph analytics (Sasaki et al. 2018).

Graphs are essential to represent or model real world phenomena or concepts such as the Internet (nodes represent computers connected to the net), molecules in chemistry (nodes represent atoms), social media networks, etc. Also, graphs are important to answer questions that are related with connected things.

1.1. Graph database concepts

This chapter presents the core concepts of the graph data model. The figure below is an example graph model with four nodes, seven relationships, properties, and labels.

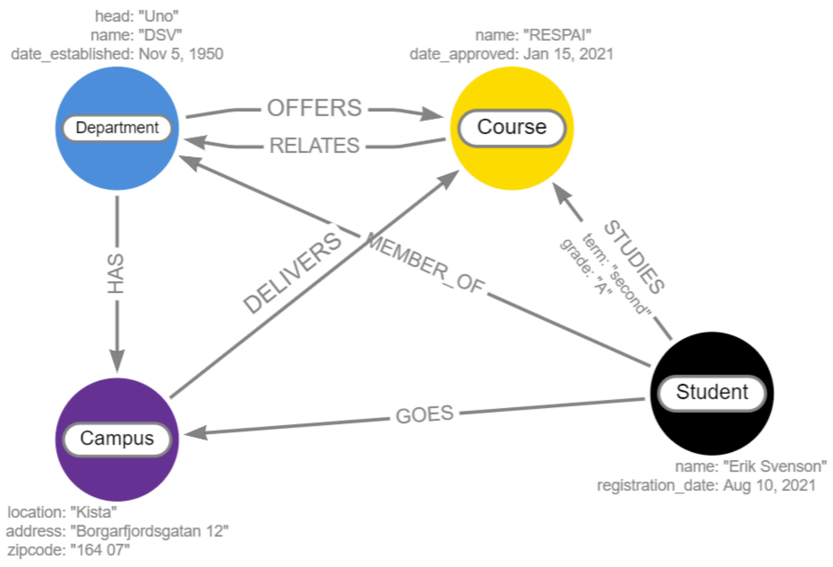


Figure 1.1 Example graph data visualization

1.2. Nodes

Nodes are discrete objects, analogues with entities in databases connected using relationships. Vertices and points are used to refer to nodes in literature. A node represents a domain entity in a graph with optional labels, property (properties), and relationship(s) with another node(s).

Nodes

- Nodes represent entities and objects; for example, in a university graph database, you may find departments, courses, campuses, and students represented as nodes in graph models.
- Data associated with nodes are represented using labels and properties. See below under Labels and Properties

1.3. Labels

Labels provide categorical information. Thus they are used to classify nodes. Nodes in a graph database may have zero or more labels. As labels are used to categorize nodes, they do not provide specific information about a node. For example, the node with the label *Course* in Figure 3.1 does not provide information about the title of a course such as - Graph Analytics, Stream Analytics, DAMI, Mathematics, and Linear Algebra.

1.4. Relationships (aka lines, edges, and links)

Graphically relationships are depicted as edges, lines or links connecting two nodes or vertices or points. Thus, relationships designate connections between nodes that are designated as source and target nodes. The relationship between source and target nodes **always has one direction** (i.e., it is not bidirectional) in Neo4j.

Note: Relationship has one direction, must have a relationship type, and optional properties.

1.4.1. Relationship type

A relationship type describes the relationship type using a verb or a verbal clause. See Figure 3.2. OFFERS and RELATES are relationship types between the *Department* and *Course* nodes. Regarding directions, the *Department* node has an outgoing relationship (OFFERS) and an incoming relationship (RELATES). In contrast, the *Course* node has an outgoing relationship (RELATES) and an incoming relationship (OFFERS).

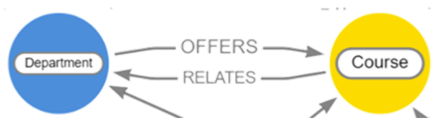


Figure 1.2 graph illustrating relationship types OFFERS and RELATES.

1.5. Properties

Properties provide specific information about a node or relationship, and properties are described as key-value pairs. Data regarding nodes and their connections are captured using properties. On the other hand, data associated with nodes are represented using labels and properties. The key-value pair part consists of keys having descriptive names that are common to zero or more nodes or relationships. In contrast, the values hold data about a specific node or relationship. In Figure 3.3, the node course has two properties with a key-value pairs *name: RESPAI and date_approved: Jan 15, 2021.* The data has a type – string, boolean, number, dates, and array of a homogenous list containing (string, boolean, number, etc.).

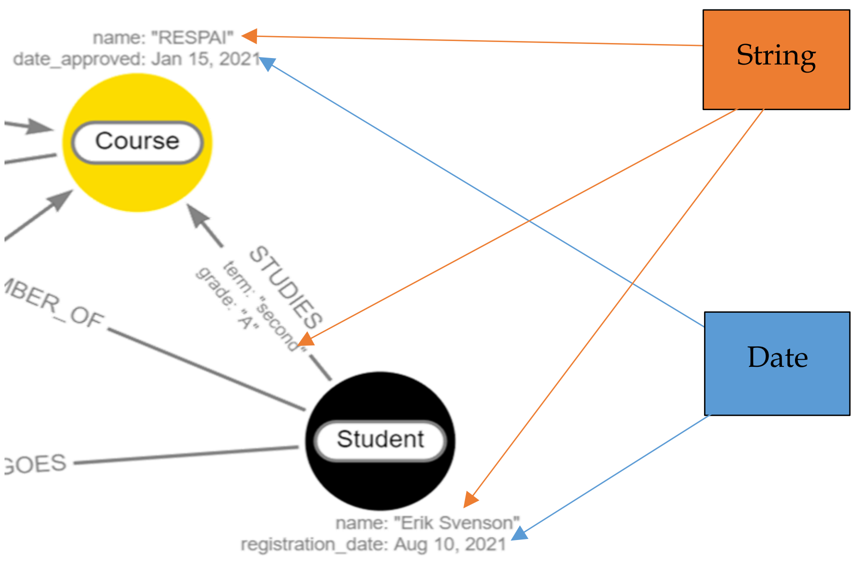


Figure 1.3 Example graph data visualization

Data types are categorized in to three parts as illustrated below:

1. ***Property types*** **-** Integer, Float, String, Boolean, Point, Date, Time, LocalTime, DateTime, LocalDateTime, and Duration.
2. ***Structural types*** **-** Node, Relationship, and Path.
3. ***Composite types*** **-** List and Map.

1.6. Naming rules and recommendations

Naming conventions

It is important to note that node labels, relationship types, and key properties are case sensitive, i.e., *email* is different from *Email*. Therefore, it is recommended to use the naming conventions described below:

Part of a Graph	Recommended naming style	Example
Node label	Camel case beginning with an uppercase	PermanentTeacher, ElectiveCourse, etc.

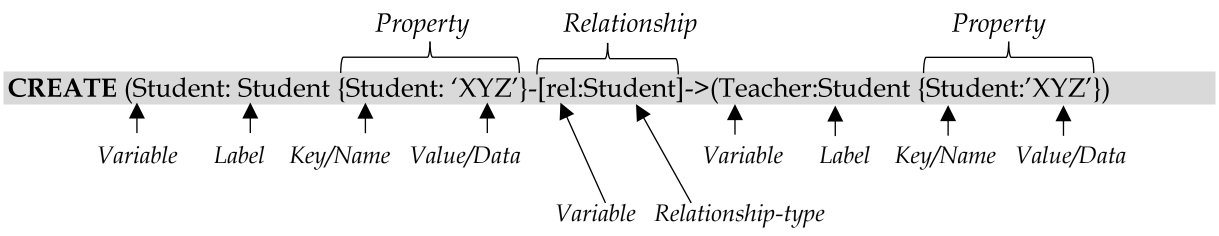
Relationship type	Uppercase with the underscore between words	MEMBER_OF, GOES, FOLLOWS, etc.
Property	Lowercase beginning with a lowercase	dateApproved, registrationDate, etc.

Naming rules

- Names should
  - begin with alphabetic characters
  - not begin with numbers
  - not contain symbols except for the underscore
- Names are case sensitive; for example, variable names, property names, label names are case sensitive
- Extra whitespaces are ignored

Scoping

- Variable names should not be reused within the same query scope
- Node labels and variables can have the same name e.g.



```
CREATE (Student: Student {Student: 'XYZ'}-[rel:Student]->(Teacher:Student {Student:'XYZ'})
```

1.7. Schema

In Neo4j, a schema refers to constraints and indexes that are used for modelling and performance benefits. However, a schema is optional in Neo4j, which means you are allowed to create nodes, relationships and properties.

Constraints

- Constraints are used to make that the graph database adheres to the rules of the domain.

Indexes

- Indexes are used to improve performances while managing graph databases, i.e., querying graph databases.

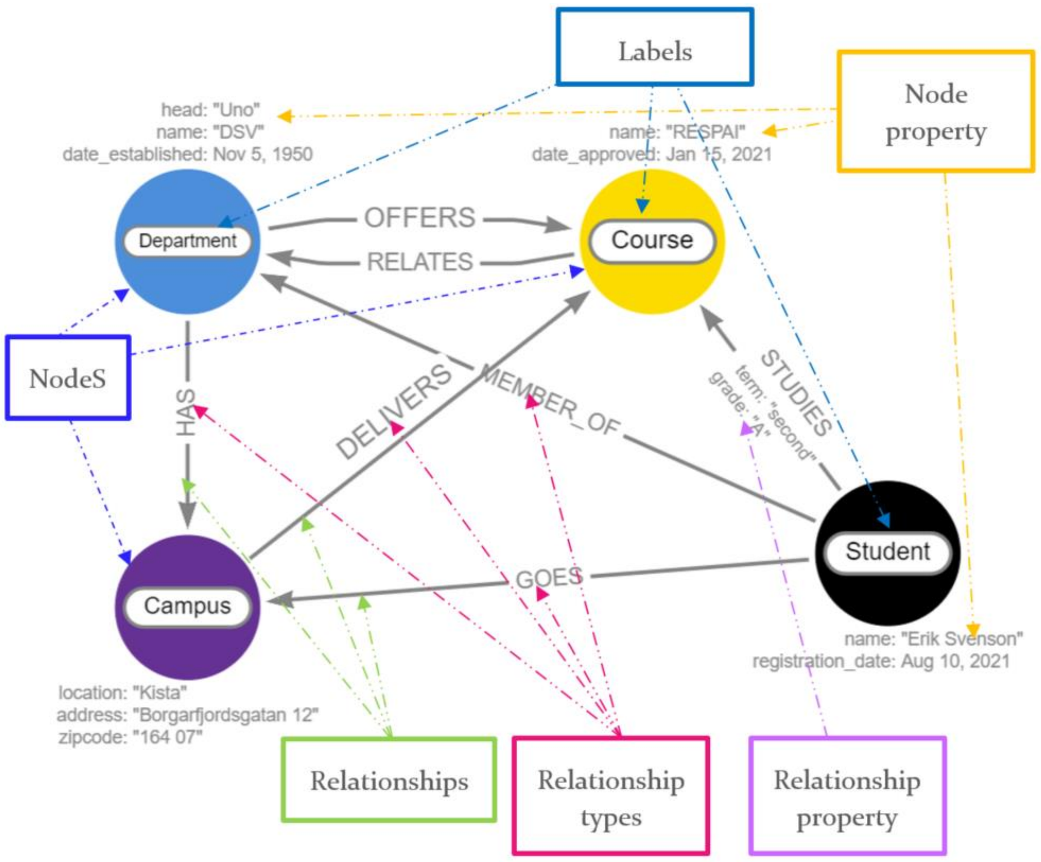


Figure 1.4 Example graph data visualization

1.8. Cypher

Cypher is a graph query language (GQL), which is currently under development for standardization alongside SQL under the category of Information Technology-Database Language (ISO/IEC 39075)[1]. In graph databases, traversal is how you query your graph database to answer questions about the graph. Thus traversing graph uses rules to visit relevant nodes and relationships and could return or be used as an input for another query. Traversing is done using Cypher. Cypher is an expressive and efficient graph query language that is used for querying, administering and updating graphs. It is inspired by well-established practices and approaches for expressive querying. For example, Cypher uses keywords that are inspired by SQL, such as ORDER BY, WHERE, UNION, etc. It also uses a SQL-like structure. Cypher borrows expressions from SPARQL[2] , which is a query language designed for an RDF (Resource Description Framework). Similarly, some of its list semantics are borrowed from Python and Haskell. Similarly, it some of its list semantics are borrowed from Python and Haskell.

1.8.1. Elements of graphs and their representations in Neo4j cypher

In this section, the components of graph data model concepts and their representations are presented. The components of graphs data include nodes, relationships, labels and

[1] https://www.iso.org/standard/76120.html
[2] https://en.wikipedia.org/wiki/SPARQL

properties as the core concepts. In Neo4j cypher, graph data components are expressed using "()", ":", "[]" and "{}".

Cypher elements	Expressed with	Meaning	Language Usage	Example
Node	()	Vertex	Noun	Graphically visualizes nodes
Label	:	Naming nodes	Noun	Provides categorical information
Relationship	[]	Edge	Verb	Follows, Loves
Property	{}	key-value pairs	Adjective (describes nouns)	Node properties include information about the node in x-y pairs

Table 1.1 graph data model concepts and their representations

1.9. Traditional DBMS vs. Graph Database

The options we have today to store connected data are relational databases, NoSQL databases, and graph databases. Relational databases and NoSQL databases lack relationships, while graph databases embrace relationships (Robinson et al., 2015).

There are semantic similarities between key terminologies used in the graph database (Neo4j) and DBMS, see Table 1.1.

Graph database concepts ex. Neo4j	DBMS
Graph	Table
Node	Rows
Property	Column
Values	Data
Constraint	Relationship
Traversal	Joins

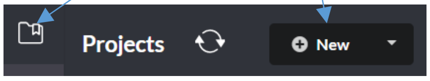
Table 1.2 semantic similarities between Neo4j terminologies and DBMS terminologies.

2. Getting started with Neo4j

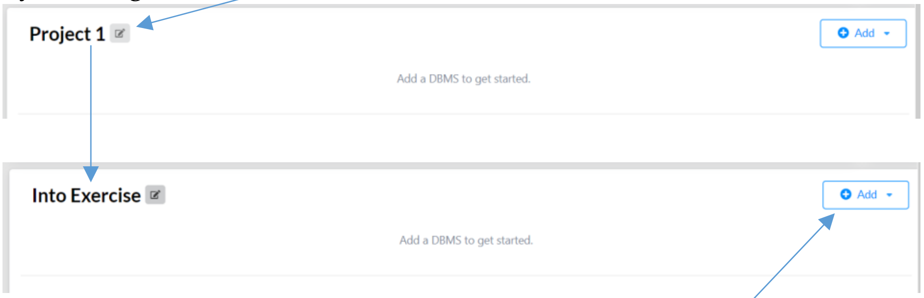
Neo4j is a type of graph database management system designed specifically for optimal traversal, storage and management of nodes and relationships. The Neo4j graph database uses a property graph approach, making it more efficient in operations runtime and traversal performance.

2.1. Create a project in Neo4j Desktop

- Start Neo4j Desktop
- Under **Projects** Click **New**



- Change the project name, possibly from *Project 1* to *Into Exercise*, as illustrated below by clicking on the **Edit** icon.



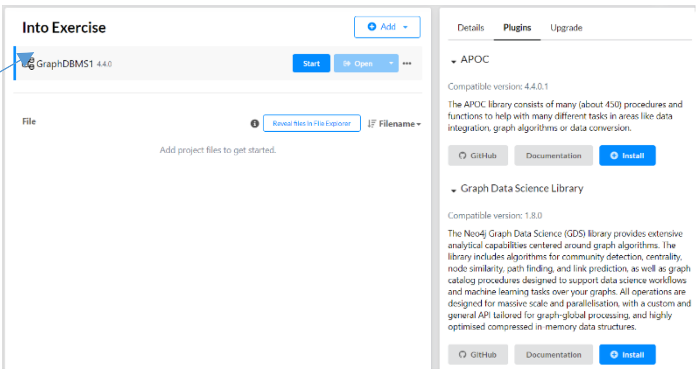
2.2. Create a database and install necessary plugins in Neo4j Desktop

Create the graph database (use version 4.4.0)

- Under the **Into Exercise** project, click on **Add**
- Select **Local DBMS** (here you have the option to connect to remote Neo4j instance, but you will not do it for this lab)
- Under **Name** enter *GraphDBMS1*
- Under **Password** enter *graph1234*
- Under **Version** select 4.4.0
- Click on **Create**

Install APOC and Graph Data Science Library plugins

- After the database is created, click on it
- Select **Plugins**
- Expand **APOC** and click on **Install**
- Expand **Data Science Library** and click on **Install**



Compatible version: 4.4.0.1

The APOC library consists of many (about 450) procedures and functions to help with many different tasks in areas like data integration, graph algorithms or data conversion.

Compatible version: 1.8.0

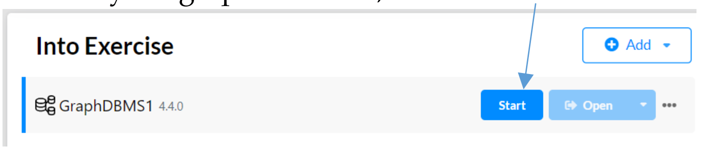
The Neo4j Graph Data Science (GDS) library provides extensive analytical capabilities centered around graph algorithms. The library includes algorithms for community detection, centrality, node similarity, path finding, and link prediction, as well as graph catalog procedures designed to support data science workflows and machine learning tasks over your graphs. All operations are designed for massive scale and parallelisation, with a custom and general API tailored for graph-global processing, and highly optimised compressed in-memory data structures.

2.3. Starting Neo4j Browser or an internet browser

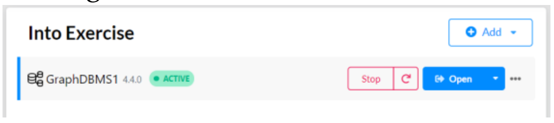
You can create and manipulate graph data in your graph database created in the previous steps using **Neo4j Browser** or an internet browser. Before you create graph data, you need to start your graph database.

Start the graph database

- To start your graph database, click on **Start**.

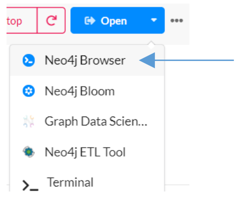


- Your graph database should be active to proceed with the next steps, as illustrated in the diagram below.



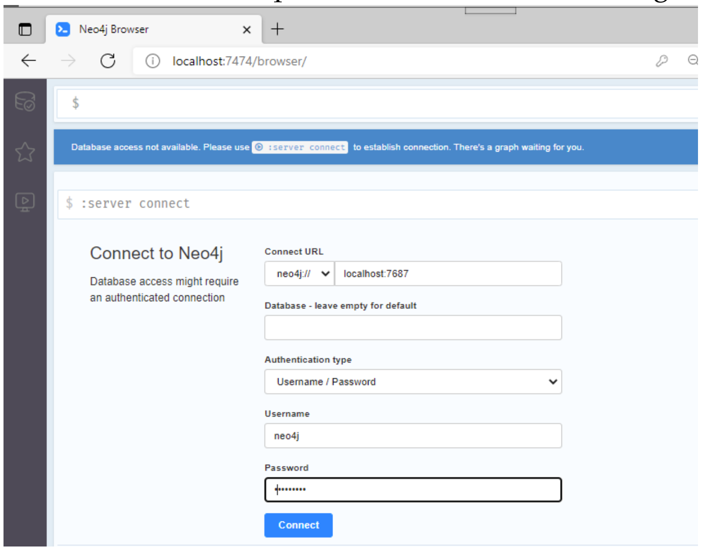
Start the Neo4j Browser

- Click on **Open**, or if you click on the drop arrow of the **Open** button, select **Neo4j Browser**



Accessing your Neo4j database using an internet browser

- Start Internet Explorer or similar internet browser
- Enter the URL http://localhost:7474/browser/ in the address bar and press **Enter** key.
- Enter username and passwords as illustrated in Figure 4.1.


Neo4j Browser

localhost:7474/browser/

$

Database access not available. Please use :server connect to establish connection. There's a graph waiting for you.

$:server connect

Connect to Neo4j

Database access might require an authenticated connection

Connect URL: neo4j:// localhost:7687

Database - leave empty for default

Authentication type: Username / Password

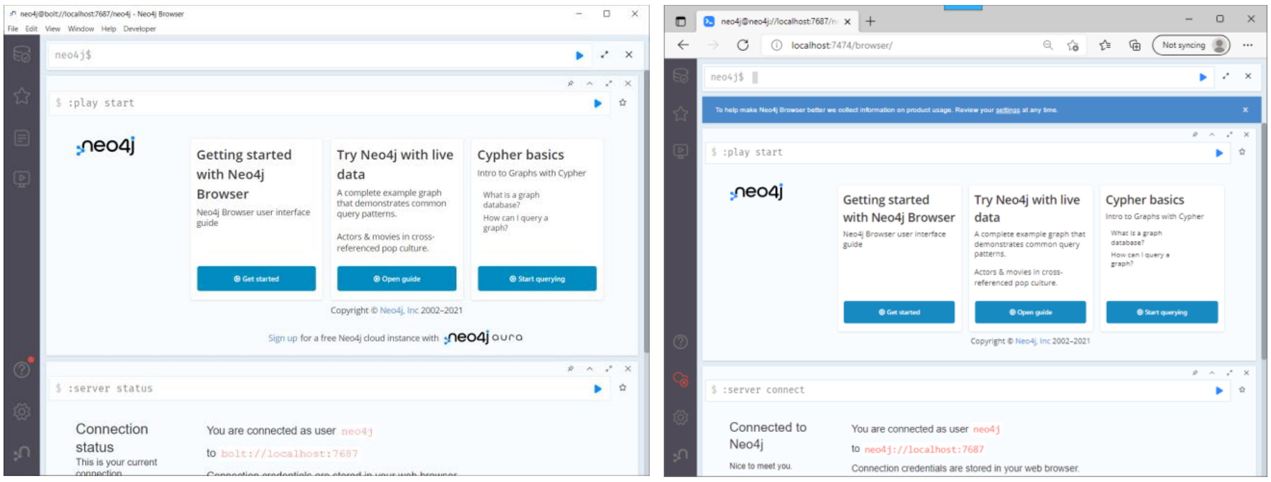
Username: neo4j

Password: ••••••••

Connect

Figure 2.1 Neo4j browser, an illustration using Internet Explorer.

- Click on **Connect**
- Now you have accessed Neo4j through the Neo4j Browser and the internet browser, you can choose either of these for creating, manipulating and querying your Neo4j graph data, as illustrated in Figure 4.2.


neo4j@bolt://localhost:7687/neo4j - Neo4j Browser
File Edit View Window Help Developer
neo4j
Get started
Open guide
Start querying
neo4j@neo4j://localhost:7687/n
localhost:7474/browser/
Not syncing

```
neo4j$
```

```
$ :play start
```

Getting started with Neo4j Browser
Neo4j Browser user interface guide

Try Neo4j with live data
A complete example graph that demonstrates common query patterns.
Actors & movies in cross-referenced pop culture.

Cypher basics
Intro to Graphs with Cypher
What is a graph database?
How can I query a graph?

Copyright © Neo4j, Inc 2002–2021

Sign up for a free Neo4j cloud instance with neo4j aura

```
$ :server status
```

Connection status
This is your current connection

You are connected as user neo4j
to bolt://localhost:7687

To help make Neo4j Browser better we collect information on product usage. Review your settings at any time.

```
$ :server connect
```

Connected to Neo4j
Nice to meet you.

You are connected as user neo4j
to neo4j://localhost:7687
Connection credentials are stored in your web browser.

Figure 2.2 Neo4j Browser on the left and Internet Explorer on the right….

3. Creating and manipulating graph data from the scratch

This section presents cypher and its clauses such as **CREATE**, **SET**, **REMOVE**, **MATCH**, **DELETE**, and **DETACH**. Cypher should be used to create, manipulate, query and visualize graph data. This section also presents different Cypher syntaxes used in Neo4j for graph data manipulation.

3.1. Creating Nodes

This section will teach you how nodes are created using the **CREATE** clause. You will use the **MATCH** clause similar to the **SELECT** clause in SQL to check if the newly created node is actually created and to create other graph data attributes.

Syntax
CREATE (variable :Label {propertyName:value})

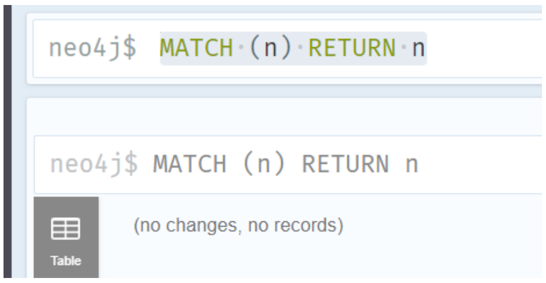
Here, the variable is optional and used to refer to the new node. Also, labels and properties are optional, yet, they enable retrieval, classification, manipulation, etc. at a later time.

Checking if there are nodes in your graph database:

- Enter the following Cypher and press **Enter** key

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

- Since you have nothing created before, you will see nothing, as illustrated below.

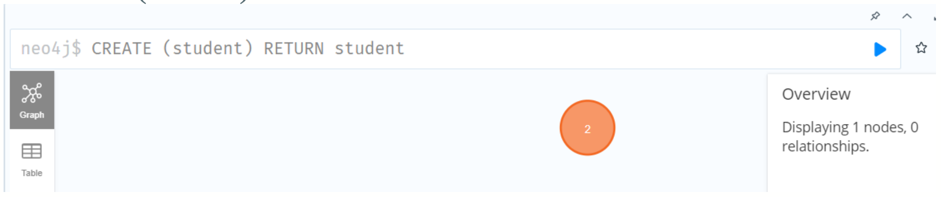

```
neo4j$ MATCH (n) RETURN n
```
```
neo4j$ MATCH (n) RETURN n
(no changes, no records)
```

Creating a node without properties and labels, run the following Cyphers one by one:

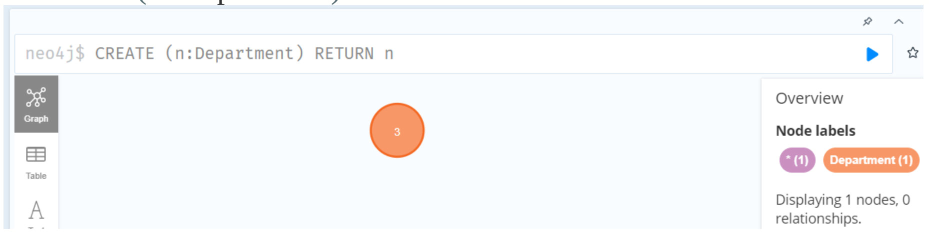
- To create a node without labels and properties.
  CREATE ()

- To create a node without labels, NOTE – student is just a variable, and it is not used in a later code.
  CREATE (student)

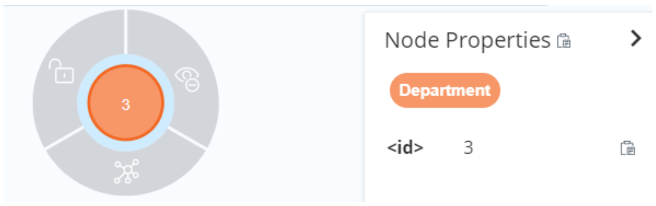
- To create a node without labels and properties, NOTE – student is just a variable, and it is used to return the node just created. The newly created node does not have a label or a property.
  CREATE (student) RETURN student


```
neo4j$ CREATE (student) RETURN student
```
Overview
Displaying 1 nodes, 0 relationships.

- To create a node with a label Department using the following Cypher.
  CREATE (n :Department) RETURN n


```
neo4j$ CREATE (n:Department) RETURN n
```
Overview
Node labels
* (1) Department (1)
Displaying 1 nodes, 0 relationships.

Note – when you click on the node or hover your mouse over it, you will see a number. This number shows you the id number of the selected node, and if the selected node is the last node, this number indicates how many nodes have been created on this machine.

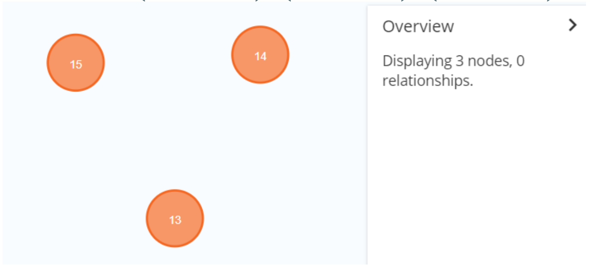

Node Properties
Department
<id> 3

Creating multiple nodes without properties and labels, run the following Cypher codes one by one:

- To create six anonymous nodes without labels and properties.
  CREATE (),(),(),(),(),()

- To create three anonymous names without labels and properties and with three unused variables (studnet1, student2, student3), NOTE – the variables are not used.
  CREATE (student1), (student2), (student3)

- To create three anonymous names without labels and properties and with three used variables (studnet1, student2, student3), NOTE – the variables are used here to return the newly created nodes.
- CREATE (student1), (student2), (student3) RETURN student1, student2, student3



Displaying 3 nodes, 0 relationships.

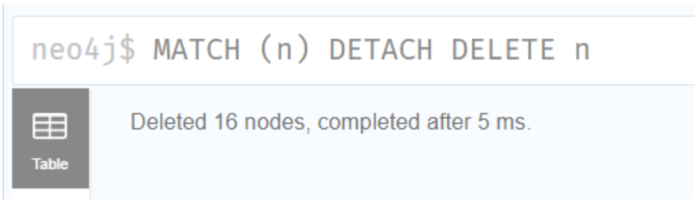
3.1.1. Deleting all nodes

- Use - MATCH (n) RETURN n ← you can use this to return all available nodes (this is similar to **SELECT * From Table1** - query)
  Or
- Use - MATCH (x) RETURN x ← x is a variable

Note – *n* and *x* are variable names, and you are free to use whatever names for variables but it is recommended to use meaningful names when you have more than one variables and when the variables are designated to hold different nodes or relationships.

Deleting all nodes from the graph database (delete all nodes created so far):

- Enter the following Cypher and press **Enter** key
  MATCH (n) DETACH DELETE n


```
neo4j$ MATCH (n) DETACH DELETE n
```

Deleted 16 nodes, completed after 5 ms.

- You can check if all nodes are deleted using: MATCH (x) RETURN x

Create nodes with a label (note that labels are case sensitive as discussed in chapter 3, which means if you create nodes with the same labels but have different cases, you will end up having different labels with the same name)

- To create a node with a label *STUDENT*

CREATE (n:STUDENT) RETURN n

- To create a node with a label *Student*
  CREATE (n:Student) RETURN n

- To create a node with a label *student*
  CREATE (n:student) RETURN n

- To create a node with a label *StuDENT*
  CREATE (n:StuDENT) RETURN n

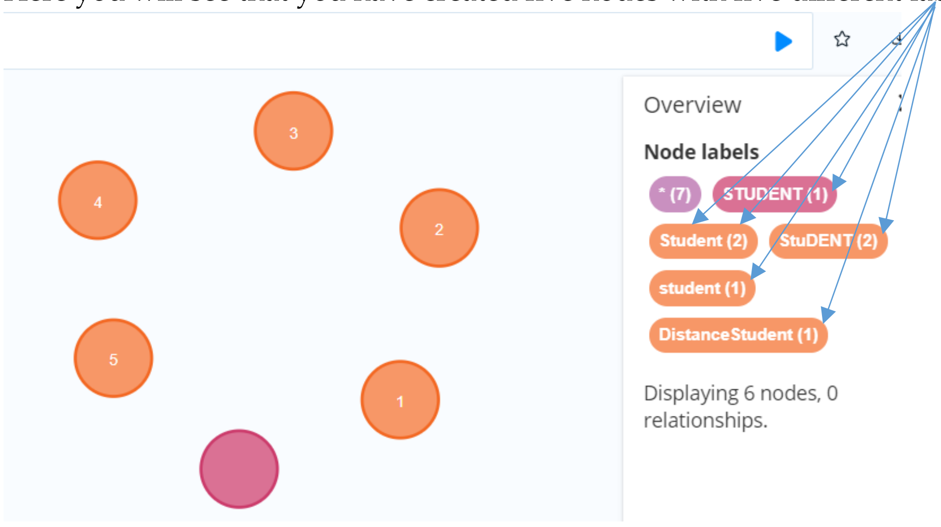
- To create a (duplicate) node with a label *StuDENT*
  CREATE (n:StuDENT) RETURN n

- Creating a node with two labels – Student and DistanceStudent
  CREATE (n:Student :DistanceStudent) RETURN n

Now, you have created six nodes with the labels STUDENT, Student, student, StuDENT, DistanceStudent, respectively. To check this, retrieve all these nodes and check how many labels are created!

Use - MATCH (n) RETURN n

- Here you will see that you have created five nodes with five different labels different



Displaying 6 nodes, 0 relationships.

Creating nodes with properties – you can use both single and double quotes (but it is recommended to use single quotes in other clauses)

- **Example Cypher codes**
- CREATE (peter: STUDENT {firstName: 'Peter',lastName: 'Tesla'})
  Or
- CREATE (peter :STUDENT {firstName: "Peter",lastName: "Tesla"})

Exercise 1

- Create two more STUDENTS (write two Cypher codes) by yourself with names, James Brown and Susan Rice.
- Create two nodes with names, Angela Markel and George Michael, using the Teacher label for both.

Exercise 1

- Create two nodes with labels Bird and Dog

3.1.2. Deleting specific node

To delete the nodes just created above, i.e., Bird and Dog nodes

- First, make sure that the nodes with labels Bird and Dog exist

```
MATCH(d:Dog) Return d        ← to check if there is a node with Dog label exists or not
MATCH(b:Bird) Return b       ← to check if there is a node with Bird label exists or not
```

- To delete the bird node:

```
MATCH (b:Bird)
DELETE b
```

- To delete the dog node:

```
MATCH (d:Dog)
DELETE d
```

To delete nodes (including anonymous nodes) using ids of these nodes. For example, if you want to delete nodes whose ids are 5 and 6 use the following cypher.

<u>**Note**</u> – if the nodes to be deleted have relationships, you will not be able to delete them. Thus you need to use **DETACH** keyword followed by the **DELETE** keyword.

3.1.3. Adding properties to nodes

Use the SET keyword to add a property to a node. Yet, you need to use the MATCH clause and optionally the WHERE clause to select the relevant node.

Syntax

```
MATCH (x:Label)
WHERE <condition>
SET x.propertyName1 = value1, x.propertyName2 = value2
```

Or

Syntax

```
MATCH (x:Label)
WHERE <condition>
SET x ={propertyName1:value1, propertyName2:value2}
```

In this case all previously created properties should be updated

Example:

Add a new property with **propertyName** *hiredDate* and assign a value *1995* to the Teacher node George Michael.

```
MATCH (x:Teacher)
WHERE x.firstName =  'George' AND x.lastName = 'Michael'
   SET x.hireDate = 1995
```

Exercise

Add a new property with **propertyName** *hiredDate* and assign a value *2001* to the Teacher node Angela Markel.

3.1.4. Updating properties of nodes

Use the SET keyword with "+=" instead of "=" to update previously created properties. However, if the property written under the SET clause is not created before for a given node, Neo4j will add the property to the node.

Syntax

```
MATCH (x:Label)
WHERE <condition>
SET x +={propertyName1:value1, propertyName2:value2}
```

Example:

Update the hiredDate value to 2010 for the node George Michael

```
MATCH (x:Teacher)
WHERE x.firstName =  'George' AND x.lastName = 'Michael'
   SET x+={hireDate : 2010}
```

3.1.5. Retrieving labels and properties of nodes

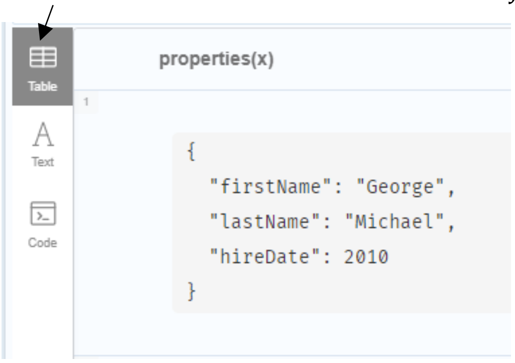
Use the MATCH clause.

Example:

Show the properties of the teacher George Michael

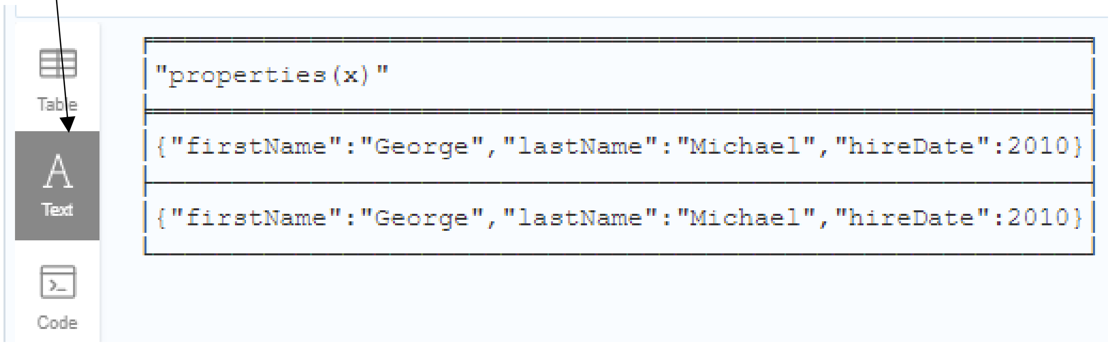
```
MATCH (x:Teacher)
WHERE x.firstName =  'George' AND x.lastName = 'Michael'
RETURN properties(x)
```

The result can be viewed in three different formats (Text, Table, and Code). You can click on: **Table** to view it in a tabular view with JSON format



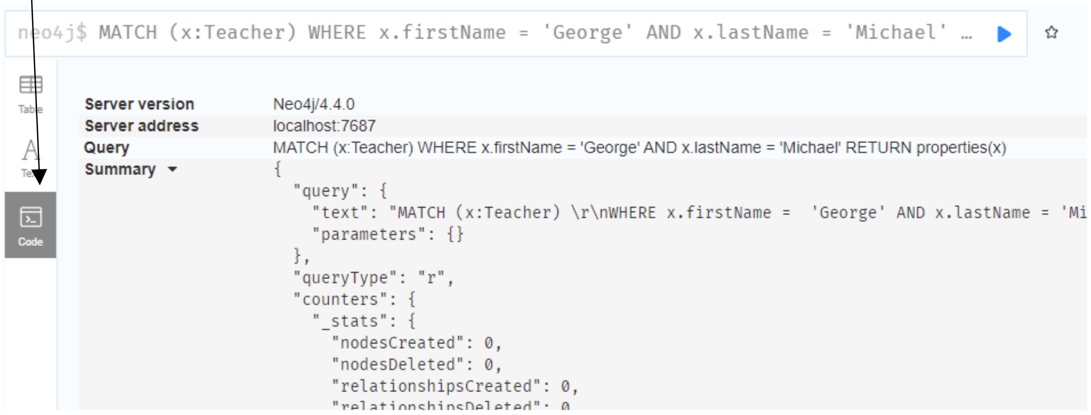
```
{
  "firstName": "George",
  "lastName": "Michael",
  "hireDate": 2010
}
```

Text to view properties in Textual format



```
"properties(x)"
{"firstName":"George","lastName":"Michael","hireDate":2010}
{"firstName":"George","lastName":"Michael","hireDate":2010}
```

Code to view summary in JSON format.



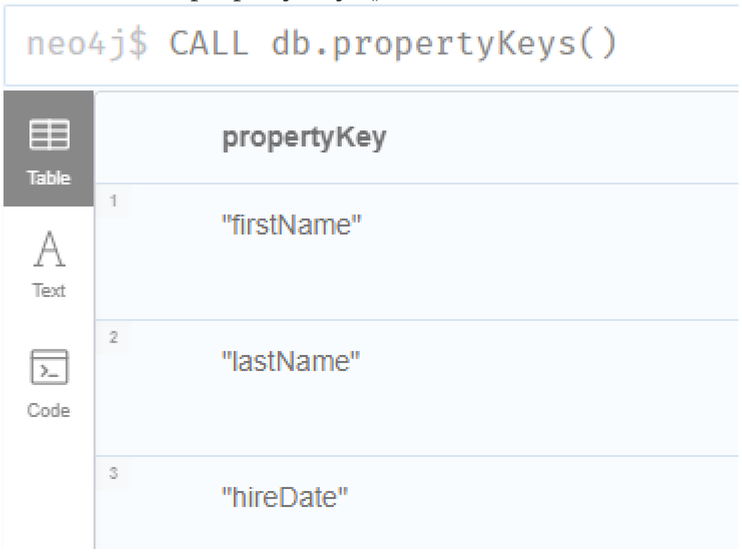
```
neo4j$ MATCH (x:Teacher) WHERE x.firstName = 'George' AND x.lastName = 'Michael' …
```

Server version Neo4j/4.4.0
Server address localhost:7687
Query MATCH (x:Teacher) WHERE x.firstName = 'George' AND x.lastName = 'Michael' RETURN properties(x)
Summary

```
{
  "query": {
    "text": "MATCH (x:Teacher) \r\nWHERE x.firstName =  'George' AND x.lastName = 'Mi
    "parameters": {}
  },
  "queryType": "r",
  "counters": {
    "_stats": {
      "nodesCreated": 0,
      "nodesDeleted": 0,
      "relationshipsCreated": 0,
```

To retrieve all property keys

- Use CALL db.propertyKeys()



```
neo4j$ CALL db.propertyKeys()
```

	propertyKey
1	"firstName"
2	"lastName"
3	"hireDate"

To view all properties of any node

MATCH (n)
RETURN properties(n)

To view all labels of all nodes

```
MATCH (n)
RETURN labels(n)
```

3.1.6. Removing properties of nodes

Use the MATCH, REMOVE, and SET clauses.

Example:

Remove the property hireDate of the teacher George Michael node.

```
MATCH (x:Teacher)
WHERE x.firstName =  'George' AND x.lastName = 'Michael'
SET x.hireDate = NULL
```

Or

```
MATCH (x:Teacher)
WHERE x.firstName =  'George' AND x.lastName = 'Michael'
REMOVE x.hireDate
```

3.1.7. Adding and removing labels of nodes

Use the MATCH, REMOVE and SET clauses to add and remove labels.

Adding labels

Example:

Add a new label, *Professor* and *SeniorLecturer*, to the Teacher *George Michael* node.

```
MATCH (x:Teacher)
WHERE x.firstName =  'George' AND x.lastName = 'Michael'
        SET x:Professor:SeniorLecturer
RETURN labels(x)
```

Removing labels
Example:

Remove the label, *Professor* and *SeniorLecturer* from the Teacher *George Michael* node.

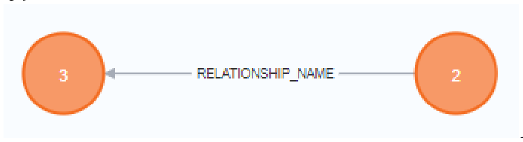
```
MATCH (x:Teacher)
WHERE x.firstName =  'George' AND x.lastName = 'Michael'
REMOVE x:Professor:SeniorLecturer
RETURN labels(x)
```

3.2. Creating relationships

3.2.1. Creating a new relationship and between two nodes

Syntax
CREATE (a) – [:RELATIONSHIP_NAME] -> (a)

In this case, if you use the Syntax as it is, it will create two anonymous nodes having a relationship named: RELATIONSHIP_NAME. But, "a" and "b" are just variable names that are not used in the cypher.



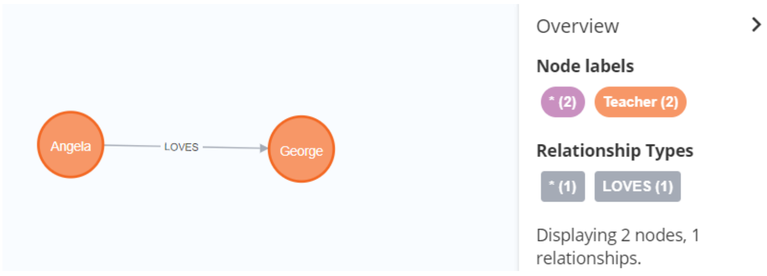
*Here the nodes have no labels, but Neo4j will assign a unique id number. Thus, if you want to create a relationship between already existing nodes, you should be more specific, so you should use the **MATCH** clause as illustrated in the following Syntax.*

Syntax
MATCH (a: node_A), (b:node_B) Create (a)-[:RELATIONSHIP_NAME]->(b)

Example:

Create a relationship with relationship type LOVES between the teachers Angela Michael and George Michael
(Ignore the warning – "*this query builds a Cartesian product between disconnected patterns…*").

```
MATCH (a: Teacher {firstName:'Angela'}), (b:Teacher {firstName: 'George'})
CREATE (a)-[:LOVES]->(b)
RETURN a,b
```



Displaying 2 nodes, 1 relationships.

Exercise

- Create a new relationship that shows that George LOVES Angela back.

To check if the new relationship is created use the following Cypher

```
MATCH (a: Teacher {firstName:'Angela'}), (b:Teacher {firstName: 'George'})
RETURN a,b
```

Exercise

- Create a new relationship that shows that George LOVES himself.

3.2.2. Creating multiple relationships at a time

You can create more than one relationships between three or more nodes at a time.

Syntax
MATCH (a: node_A), (b:node_B), (c:node_C) Create (a)-[:RELATIONSHIP_NAME1]->(b)<-[:RELATIONSHIP_NAME2]-(c)

Example

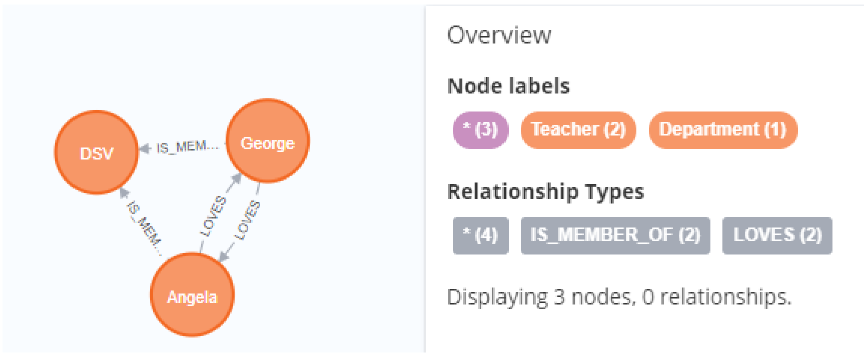
Now let us create a new node with a Label Department and propertIES departmentName = DSV and location = 'Kista'. Then, create two relationships between the two teachers (Angela and George) and the department DSV having a relationship type IS_MEMBER_OF.

To create the department node use:

```
CREATE (d:Department {departmentName:'DSV', location :'Kista'})
RETURN d
```

To create the relationship between teachers and the department:

```
MATCH (a: Teacher {firstName:'Angela'}), (b:Teacher {firstName: 'George'}), (c:Department
{departmentName:'DSV'})
Create (a)-[:IS_MEMBER_OF]->(c)<-[ :IS_MEMBER_OF]-(b)
RETURN a,b,c
```



Displaying 3 nodes, 0 relationships.

3.2.3. Creating relationships with properties

In this section, you will learn how a relationship with one or more property is created.

Syntax
MATCH (a: node_A), (b:node_B) Create (a)-[:RELATIONSHIP_NAME1{propertyName1:value1, propertyName2:value2}]->(b)

Exercise

- Create a new node with a label University and a property univeristyName = 'SU'.

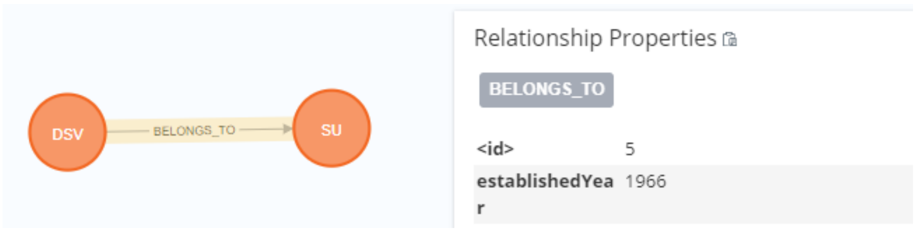
Example:

Create a relationship between the University SU and the Department DSV showing DSV BELONGS_TO SU with a property establishedYear = 1966

The solution is:

```
MATCH (a: University {universityName:'SU'}), (b:Department
{departmentName:'DSV'})
Create (b)-[:BELONGS_TO {establishedYear:1966}]->(a)
RETURN a,b
```

The relationship type is shown when you click on the relationship as illustrated below.



Relationship Properties

BELONGS_TO

<id>	5
establishedYear	1966

Exercise

- Create a new University node with universityName = 'KTH'
- Create a new relationship between George Michael and the Univerity KTH showing that George is a PART_TIME_OF KTH.

3.2.4. Adding properties to relationships with properties

It is possible to add properties to relationships but you need to use the MATCH clause to specify which nodes and relationships you want to add properties to.

Syntax - for a relationship type rel

SET rel.propertyName1 = value1 ← to set one property

SET rel.propertyName1 = value1, rel.propertyName2 = value2 ← to set two properties
The set properties using the equal sign, "=", previously created values will be removed if they are not included in the new one

SET rel = {propertyName1:value1,propertyName2:value2} ← set properties
If you are using this style you need to include all properties otherwise previously created values will be removed.

SET rel += {propertyName1:value1 } ← this is used to update properties
If you are using this style, you will be able to update properties, yet if the property does not exit it will be created.

3.2.5. Removing properties from relationships

It is possible to remove properties of relationships but you need to use the **MATCH** clause to specify which nodes and relationships you want to remove properties to.

Syntax - for a relationship type rel1 and rel2

```
REMOVE rel.propertyName1                        ← to remove one property

SE rel.propertyName1  = NULL                    ← to remove one property
```

4. Creating and manipulating graph data using MERGE clause

If you use the **CREATE** clause several times for creating:

- Nodes – duplicates will be created
- Labels – the node is not updated
- Relationships – duplicates will be created
- Properties – for node properties and relationship properties it is updated with new values

If you use the **MERGE** clause instead of the **CREATE** clause, it enables you to create unique nodes and relationships. For example, you use a **MERGE** instead of **CREATE** for a node, it first matches and checks if there is any node with the nodes specification, and if there is no node, it will create a new one. The use of the **MERGE** clause is similar to the use of **MATCH** and **CREATE** combined, where you will be able to create something new if the **MATCH** returns a value.

NOTE – It is recommended to create indexes as they boost the performance when **MERGE** is used. For example, when you want to create a node using **MERGE**, **MERGE** will scan through all nodes, and if your dataset is large enough to cause performance issues, it is recommended to create indexes. You will learn about indexes and constraints in the next chapter.

4.1.1. Creating nodes using MERGE

Syntax
MERGE (variable :Label {propertyName:value})

Example: Merging a node that has already been created – you have created a department node with departmentName = 'DSV' and location = 'Kista'.What will happen if you merge it as follows?

Step 1 - First check if the node exists using MATCH

```
MATCH (d:Department {departmentName:'DSV', location :'Kista'})
RETURN d
```

Step 2 – Use the MERGE clause to create the node

```
MERGE (d:Department {departmentName:'DSV', location :'Kista'})
RETURN d
```

It will return the same information as with the **MATCH** clause because the node already exists.

Step 3 – Use the MERGE clause to create the node by omitting the departmentName while keeping the location property

```
MERGE (d:Department { location :'Kista'})
RETURN d
```

It will still return the same information as with the MATCH clause because the node already exits.

Exercise

- What will happen if you use the following Cypher?
  ```
  CREATE (d:Department { location :'Kista'})
  RETURN d
  ```

5. Constraints and indexes

5.1. Constraints

Similar to RDBMS, constraints should be enforced to graph databases such as uniqueness and existence constraints. However, there is no primary key constraint in graph databases, unlike relational databases. Examples of constraints in Neo4j are:

- **Uniqueness constraints** ensure that all nodes have a unique value for a property on which the constraint is enforced.
- **Existence constraint** (only applicable in Enterprise Edition and latest Desktop versions) ensures that a certain property exists for all new nodes, and if you want to add existence constraints, the constraint should not be violated by existing nodes.
- **Node key** (only applicable in Enterprise Edition and latest Desktop versions) ensures that a set of values of the combined properties of a given node is unique, thus based on the combined properties of nodes, the uniqueness of nodes is ensured.

For this example, create a new node with a label Course and a property called title with a value 'DBMS'.

```
CREATE (c :Course {title:'DBMS'})
```

5.1.1. Example: - Unique constraint

- Create a unique constraint for the Course DBMS.

```
CREATE CONSTRAINT UniqueCourseConstraint ON (c :Course) Assert c. title IS UNIQUE
```

Exercise
What will happen if you use **Cypher A** and **Cypher B**? Explain the difference?

Cypher A

```
CREATE (c :Course {title:'DBMS'})
```

Cypher B

```
MERGE (c :Course {title:'DBMS'})
```

5.1.2. Example: - Existence constraint

- Create a unique constraint for the Course DBMS.

```
CREATE CONSTRAINT ExistTitle ON (c :Course) Assert exists(c. title)
```

Exercise
What will happen if you use **Cypher A**? Explain.

Cypher A

```
CREATE (c :Course {approvedYear:1990})
```

Retrieving all available constraints

Use `CALL db.constraints()`

Deleting constraints

```
DROP CONSTRAINT constraing_name
```

5.1.3. Example: - Node key constraint

- Create a node key constraint for the node Teachers using the properties firstName and lastName.

```
CREATE CONSTRAINT constraint_node_key_name FOR (t:Teacher) REQUIRE
(t.firstName, t.lastName) IS NODE KEY
```

Exercise

- What will happen if you use **Cypher A** and **Cypher B**? Explain.

Cypher A

```
CREATE (:Teacher {firstName: 'Angela', lastName: 'Markel'})
```

Cypher B

```
MERGE (:Teacher {firstName: 'Angela', lastName: 'Markel'})
```

5.2. Indexes

Indexes are redundant copies of data of a database for increasing search performance. Indexes are updated by the database once they are created when the graph data is changed. This is possible through additional cost as there is an extra write, so you need to decide what to index. An index on a single property is referred to as a single-property index, while an index composed of more than one property is referred to as a composite index.

Single-property indexes are used for:

- Range comparison i.e., >,<, >=, <=
- Equality checks =
- String comparisons where **CONTAINS**, **STARTS WITH**, and **ENDS WITH** are checked
- Existence checks using **exists ()**
- List membership check using **IN**

Retrieving all available indexes

Use `CALL db.indexes()`

Deleting constraints

```
DROP INDEX index_name
```

6. References

Sasaki, B. M., Chao, J., & Howard, R. (2018). Graph databases for beginners. *Neo4j*.

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".

https://neo4j.com/docs/bloom-user-guide/current/

https://neo4j.com/docs/cypher-manual/current/