<https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/>

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

<https://neo4j.com/docs/cypher-refcard/current/>

**Introduction to Neo4j Online Course and Tutorial**

**About the Course**

This online course is a collection of lessons and tutorials which help you build a good starting knowledge of graph databases in less than a day. It also teaches the core functionality of the Neo4j graph database. You will learn the foundation knowledge required to start building applications with Neo4j, including how to read and write Cypher queries.

**Course Outline**

* **Introduction to Graph Databases ✅**
* **Introduction to Neo4j ✅**
* **Setting up your Development Environment Tutorial ✅**
* **Introduction to Cypher ✅**
* **Getting More out of Queries ✅**
* **Creating Nodes and Relationships** **✅**
* **Getting More out of Neo4j ✅**

### [Introduction to Graph Databases](https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/part-1/)

* Describe what a graph database is.
* Describe how to model relational data in a property graph model.

Estimated time: 30 minutes

### [Introduction to Neo4j](https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/part-2/)

* Describe the Neo4j Graph Platform and its components.
* Describe the features and benefits of Neo4j.

Estimated time: 30 minutes

### [Setting Up Your Development Environment](https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/part-3/)

* Set up your development environment for performing the hands-on exercises of this course which is one of:
  + Neo4j Sandbox
  + Neo4j Desktop

Estimated time: 30 minutes

### [Introduction to Cypher](https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/part-4/)

* Use MATCH to retrieve nodes from the graph.
* Use MATCH to retrieve relationships from the graph.
* Use MATCH to retrieve properties from the graph.

Estimated time: 60 minutes

### [Getting More Out of Queries](https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/part-5/)

* Use the WHERE clause for queries.
* Control query processing.
* Control how results are returned.
* Work with Cypher dates and lists.

Estimated time: 90 minutes

### [Creating Nodes and Relationships](https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/part-6/)

* Create, update, and delete nodes and properties of nodes.
* Create, update, and delete relationships and properties of relationships.
* Merge data in the graph.

Estimated time: 90 minutes

### [Getting More Out of Neo4j](https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/part-7/)

* Use parameters.
* Define constraints in the graph.
* Profile and monitor query execution.
* Define indexes in the graph.
* Import relational data into the graph.

Estimated time: 90 minutes

### [Summary](https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/part-8/)

* Quiz results.
* Resources to learn more.
* Course feedback.

MODULE 1

Today’s business and user requirements demand applications that connect more and more of the world’s data, yet still expect high levels of performance and data reliability. Many applications of the future will be built using graph databases like Neo4j.

In this video, you will learn how the need for graph databases has evolved.

[ANA]

NEO4J is a transactional NOSQL database which still has the ACID properties

INTUITIVENESS --- SPEED --- AGILITY

**Cypher** is a declarative QUERY LANGUAGE (similar to SQL) that is optimized for GRAPHS.

### What Is a graph database?

A graph database is an online database management system with Create, Read, Update and Delete (CRUD) operations working on a graph data model. Graph databases are generally built for use with online transaction processing (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

Unlike other databases, relationships take first priority in graph databases. This means your application doesn’t have to infer data connections using foreign keys or out-of-band processing, such as MapReduce.

By assembling the simple abstractions of nodes and relationships into connected structures, graph databases enable us to build sophisticated models that map closely to our problem domain.

### The case for graph databases

The biggest value that graphs bring to the development stack is their ability to store relationships and connections as first-class entities.

For instance, the early adopters of graph technology reimagined their businesses around the value of data relationships. These companies have now become industry leaders: LinkedIn, Google, Facebook and PayPal.

As pioneers in graph technology, each of these enterprises had to build their own graph database from scratch. Fortunately for today’s developers, that’s no longer the case, as graph database technology is now available off the shelf.

In this video, you will learn how graph databases help you to model real-world data that needs to be connected as well as how Neo4j is used to solve real problems facing enterprises today.

### What is a graph?

A graph is composed of two elements: **nodes** and **relationships**.

Each node represents an entity (a person, place, thing, category or other piece of data). With Neo4j, nodes can have **labels** that are used to define types for nodes. For example, a Location node is a node with the label Location. That same node can also have a label, Residence. Another Location node can also have a label, Business. A label can be used to group nodes of the same type. For example, you may want to retrieve all of the Business nodes.

Each relationship represents how two nodes are connected. For example, the two nodes Person and Location, might have the relationship LIVES\_AT pointing from a Person node to Location node. A relationship represents the verb or action between two entities. The MARRIED relationship is defined from one Person node to another Person node. Although the relationship is defined as directional, it can be queried in a non-directional manner. That is, you can query if two Person nodes have a MARRIED relationship, regardless of the direction of the relationship. For some data models, the direction of the relationship is significant. For example, in Facebook, using the KNOWS relationship is used to indicate which Person invited the other Person to be a friend.

This enables you to closely align data and connections in the graph to your real-world application. For example, a *Person* node might have a property, *name* and a *Location* node might have a property, *address*. In addition, a relationship, *MARRIED* , might have a property, *since*.

In this video, you will learn how to model property graphs containing nodes and relationships and how Cypher is used to access a graph database.

### Modeling relational to graph

Many applications’ data is modeled as relational data. There are some similarities between a relational model and a graph model:

| **Relational** | **Graph** |
| --- | --- |
| Rows | Nodes |
| Joins | Relationships |
| Table names | Labels |
| Columns | Properties |

But, there are some ways in which the relational model differs from the graph model:

| **Relational** | **Graph** |
| --- | --- |
| Each column must have a field value. | Nodes with the same label aren’t required to have the same set of properties. |
| Joins are calculated at query time. | Relationships are stored on disk when they are created. |
| A row can belong to one table. | A node can have many labels. |

#### **Run-time behavior: RDBMS vs graph**

How data is retrieved is very different between an RDBMS and a graph database:

#### **How we model: RDBMS vs graph**

How you model data from relational vs graph differs:

| **Relational** | **Graph** |
| --- | --- |
| Try and get the schema defined and then make minimal changes to it after that. | It’s common for the schema to evolve with the application. |
| More abstract focus when modeling i.e. focus on classes rather than objects. | Common to use actual data items when modeling. |

If we were modeling a football transfers graph in relational and graph databases these diagrams show what common approaches might look like.

Here is the relational model:

### How does Neo4j support the property graph model?

* Neo4j is a **Database** – use it to reliably **store information** and **find it later**.
* Neo4j’s data model is a **Graph**, in particular a **Property Graph**.
* **Cypher** is Neo4j’s graph query language (**SQL for graphs!**).
* Cypher is a declarative query language: it describes **what** you are interested in, not **how** it is acquired.
* Cypher is meant to be very **readable** and **expressive**.

### Summary

You should now be able to:

* Describe what a graph database is.
* Describe some common use cases for using a graph database.
* Describe how real-world scenarios are modeled as a graph.

MODULE 2

# **Introduction to Neo4j**

### About this module

The Neo4j Graph Platform enables developers to create applications that are best architected as graph-powered systems that are built upon the rich connectedness of data.

At the end of this module, you should be able to:

* Describe the components and benefits of the Neo4j Graph Platform.

### Neo4j Graph Platform

The Neo4j Graph Platform includes components that enable you to develop your graph-enabled application. To better understand the Neo4j Graph Platform, you will learn about these components and the benefits they provide.

#### **Neo4j Database**

The heart of the Neo4j Graph Platform is the Neo4j Database. The Neo4j Graph Platform includes out-of-the-box tooling that enables you to access graphs in Neo4j Databases. In addition, Neo4j provides APIs and drivers that enable you to create applications and custom tooling for accessing and visualizing graphs.

#### **Neo4j Database: Index-free adjacency**

With index free adjacency, when a node or relationship is written to the database, it is stored in the database as connected and any subsequent access to the data is done using pointer navigation which is very fast. Since Neo4j is a native graph database (i.e. it has a graph as its core data model), it supports very large graphs where connected data can be traversed in constant time without the need for an index.

***Neo4j Database: ACID (Atomic, Consistent, Isolated, Durable)***

Transactionality is very important for robust applications that require an ACID (atomicity, consistency, isolation, and durability) guarantees for their data. If a relationship between nodes is created, not only is the relationship created, but the nodes are updated as connected. All of these updates to the database must all succeed or fail.

***Clusters***

Neo4j supports clusters that provide high availability, scalability for read access to the data, and failover which is important to many enterprises.

#### **Graph engine**

The Neo4j graph engine is used to interpret Cypher statements and also executes kernel-level code to store and retrieve data, whether it is on disk, or cached in memory. The graph engine has been improved with every release of Neo4j to provide the most efficient access to an application’s graph data. There are many ways that you can tune the performance of the engine to suit your particular application needs.

#### **Language and driver support**

Because Neo4j is open source, you can delve into the details of how the Neo4j Database is accessed, but most developers simply use Neo4j without needing a deeper understanding of the underlying code. Neo4j provides a full stack that implements all levels of access to the database and clustering layer where you can use our published APIs. The language used for querying the Neo4j database is Cypher, an open source language.

In addition, Neo4j supports Java, JavaScript, Python, C#, and Go drivers out of the box that use Neo4j’s bolt protocol for binary access to the database layer. Bolt is an efficient binary protocol that compresses data sent over the wire as well as encrypting the data. For example, you can write a Java application that uses the Bolt driver to access the Neo4j database, and the application may use other packages that allow data integration between Neo4j and other data stores or uses as common framework such as spring.

It is also possible for you to develop your own server-side extensions in Java that access the data in the database directly without using Cypher. The Neo4j community has developed drivers for a number of languages including Ruby, PHP, and R. You can also extend the functionality of Neo4j by creating user defined functions and procedures that are callable from Cypher.

#### **Libraries**

Neo4j has a published, open source Cypher library, Awesome Procedures on Cypher (APOC) that contain many useful procedures you can call from Cypher. Another Cypher library is the Graph Algorithms library, shown here, that can help you to analyze data in your graphs. Graph analytics are important because with Neo4j, the technology can expose questions about the data that you never thought to ask. And finally, you can use the GraphQL library (tree-based subset of a graph) to access a Neo4j Database. These libraries are available as plug-ins to your Neo4j development environment, but there are many other libraries that have been written by users for accessing Neo4j.

Tools

In a development environment, you will use the Neo4j Browser or a Web browser to access data and test your Cypher statements, most of which will be used as part of your application code. Neo4j Browser is an application that uses the JavaScript Bolt driver to access the graph engine of the Neo4j database server. Neo4j also has a new tool called **Bloom** that enables you to visualize a graph without knowing much about Cypher. In addition, there are many tools for importing and exporting data between flat files and a Neo4j Database, as well as an ETL tool.

In this video, you can see how Neo4j Bloom can be used to examine and modify a Graph, even when you know very little about Cypher:

#### **Whiteboard modeling**

With a property graph model, it is very easy to collaborate with colleagues to come up with a whiteboard model of your data that is easy to understand and easy modify. You then use the model to create the nodes, relationships, labels, and properties you will use for your Neo4j data. Even after the graph has been defined and populated with data, it is easy to modify the graph as your application needs change.

Neo4j Graph Platform architecture

Here is the big picture of the Neo4j Graph Platform. The Neo4j Database provides support for graph transactions and analytics. Developers use the Neo4j Desktop, along with Neo4j Browser to develop graphs and test them, as well as implement their applications in a number of languages using supported drivers, tools and APIs. Administrators use tools to manage and monitor Neo4j Databases and clusters. Business users use out-of-the box graph visualization tools or they use custom tools. Data analysts and scientists use the analytics capabilities in the Graph Algorithm libraries or use custom libraries to understand and report findings to the enterprise. Applications can also integrate with existing databases (SQL or NoSQL), layering Neo4j on top of them to provide rich, graph-enabled access to the data.

MODULE 3

# **Setting Up Your Development Environment**

### About this module

As a developer, you will create Neo4j Databases, add and update data in them, and query the data. When you learn to use Neo4j as a developer, you have three options⎼ Neo4j Desktop, Neo4j Aura, or Neo4j Sandbox. In this module you will learn how to use each of these development environments and select the option that is best for your needs while you are learning about Neo4j.

Many graph-enabled applications have been developed and deployed using Neo4j’s Community Edition (free). If your enterprise requires production features such as failover, clustering, monitoring, advanced access control, secure routing, etc. you will use Neo4j’s Enterprise Edition, or you can use Neo4j Aura which is the cloud instance of a Neo4j Enterprise Edition Database.

At the end of this module, you should be able to:

* Determine the development environment that is best for you:
  + Install and start using the Neo4j Desktop.
  + Create a Neo4j Database instance in the cloud with Neo4j Aura.
  + Create a Neo4j Sandbox for learning Neo4j.
* Start using Neo4j Browser.

Neo4j Desktop

The Neo4j Desktop includes the Neo4j Database server which has a graph engine and kernel so that Cypher statements can be executed to access a database on your system. It includes an application called Neo4j Browser. Neo4j Browser enables you to access a Neo4j database using Cypher. You can also call built-in procedures that communicate with the database server. There are a number of additional libraries and drivers for accessing the Neo4j database from Cypher or from another programming language that you can install in your development environment. If you are looking to use your system for application development and you want to be able to create multiple Neo4j databases on your machine, you should consider downloading the Neo4j Desktop (free download). The Neo4j Desktop runs on OS X, Linux, and Windows.

Neo4j Aura

Neo4j Aura enables you to create a Neo4j Database instance in the Cloud using a monthly subscription model. The amount per month depends on the amount of memory required for the database. This frees you from needing to install Neo4j on your system. Once you create a Neo4j Database at the [Neo4j Aura site](https://console.neo4j.io/), it will be managed by Neo4j. Backups are done automatically for you and the database is available 24X7. In addition, the Neo4j will ensure that the database instance is always up-to-date with the latest version of Neo4j.

Neo4j Sandbox

The Neo4j Sandbox is another way that you can begin development with Neo4j. It is a temporary, cloud-based instance of a Neo4j Server with its associated graph that you can access from any Web browser. The database in a Sandbox may be blank or it may be pre-populated. It is started automatically for you when you create the Sandbox.

By default, the Neo4j Sandbox is available for three days, but you can extend it for up to 10 days. If you do not want to install Neo4j Desktop on your system, consider creating a Neo4j Sandbox. You must make sure that you extend your lease of the Sandbox, otherwise you will lose your graph and any saved Cypher scripts you have created in the Sandbox. However, you can use Neo4j Browser Sync to save Cypher scripts from your Sandbox. We recommend you use the Neo4j Desktop or Neo4j Aura for a real development project. The Sandbox is intended as a temporary environment or for learning about the features of Neo4j as well as specific graph use-cases.

### Steps for setting up your development environment for this training

**If you are using Neo4j Desktop**:

1. Install Neo4j Desktop.
2. In a project, create a local graph (database).
3. Start the database.
4. Click the Neo4j Browser application.

**If you are using Neo4j Aura**:

1. Create a 1G Neo4j Database in your Neo4j Aura account that is already started for you.
2. Click the link to access Neo4j Browser.

**If you are using Neo4j Sandbox**:

1. Create a Neo4j Sandbox (blank Sandbox). The Sandbox has a db already started for you.
2. Click the link to access Neo4j Browser.

### ****Guided Exercise: Getting Started with Neo4j Desktop****

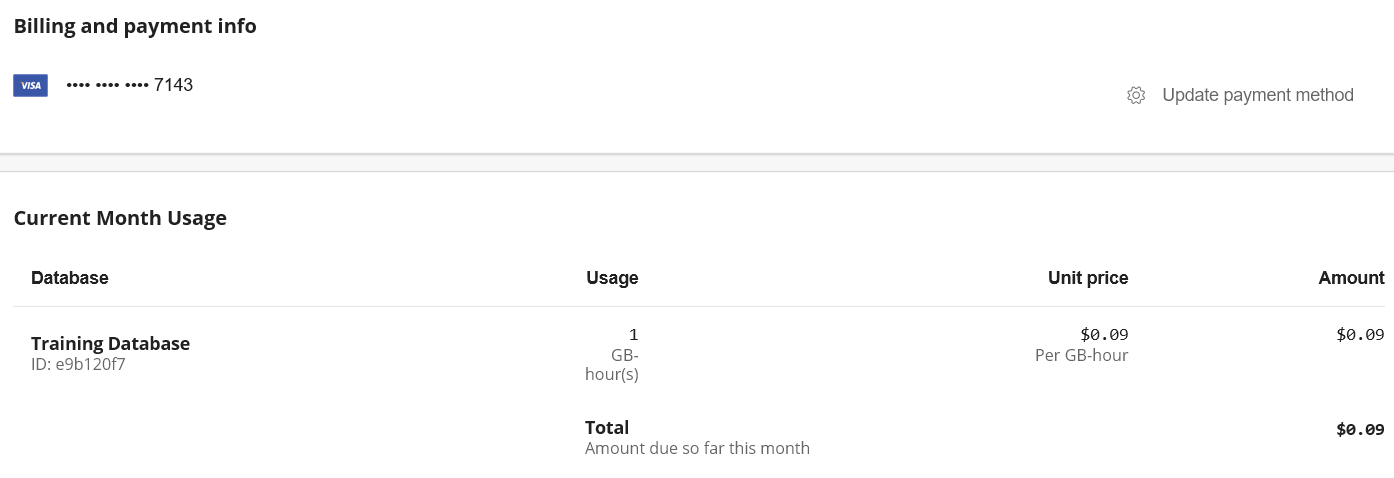
In order to perform the hands-on exercises for this training, you must install Neo4j Desktop, create a Database in Neo4j Aura, or create a Neo4j Sandbox.

If you want to download and install Neo4j Desktop on your system, follow along with one of these videos to download, install and get started using Neo4j Desktop. If you will be using Neo4j Desktop in your development environment, you can follow the steps in the video to create a TestMovies project with its corresponding Movies database.

[ANA] console.neo4j.io

wP5EmqOLSUiMGqF4NS6gCFLXEyajqbt8BfeOaq3pwjE

**DELETED!!!!!**



If you do not plan on using your database we recommend [deleting the database to avoid such charges](https://aura.support.neo4j.com/hc/en-us/articles/360045591454-Deleting-your-Neo4j-Aura-database).

No, you will only be billed for each hour, or fraction thereof, during which your database(s) existed. We display the monthly price, based on 24-hour days and 30-day months, merely as a guide to help you budget your use of the service.

Running continuously for a 30 day month will cost you $518.40

**Guided Exercise: Creating a Neo4j Sandbox**

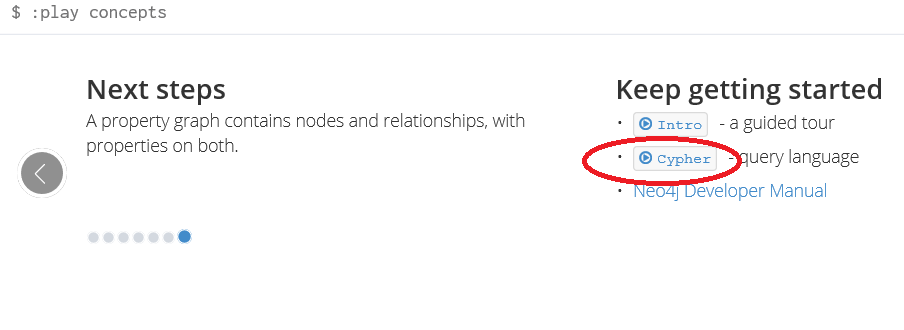
In order to perform the hands-on exercises for this training, you must install Neo4j Desktop, create a Database in Neo4j Aura, or create a Neo4j Sandbox.

If you want to run the Neo4j Database server in the cloud using a Neo4j Sandbox, follow along with this video to create a Neo4j Sandbox for training purposes. For this course, you should select the Blank Sandbox.

[ANA] sandbox.neo4j.com

IN the browser, click in START LEARNING, and advance through the guide!!

Try the CYPHER browser guide to learn simple statements:

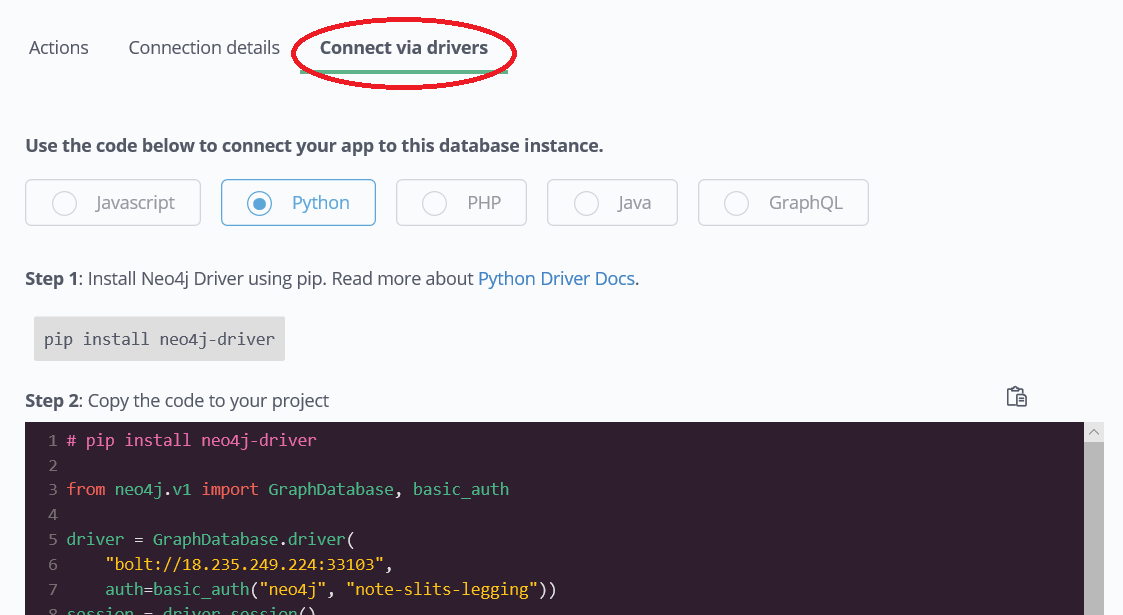


[ANA] click the example code (in the guide) and then execute!

CREATE (ee:Person { name: "Emil", from: "Sweden", klout: 99 })

CREATE (ee:Person { name: "Ana", from: "Brazil", klout: 90 })

To connect to the sandbox via PYTHON:



### Using Neo4j Browser

Neo4j Browser is a tool that enables you to access a Neo4j Database by executing Cypher statements to create or update data in the graph and to query the graph to return data. The data returned is typically visualized as nodes and relationships in a graph, but can also be displayed as tables. In addition to executing Cypher statements, you can execute a number of system calls that are related to the database being accessed by the Browser. For example, you can retrieve the list of queries that are currently running in the server.

There are two ways that you can use Neo4j Browser functionality:

* Open the Neo4j Browser application from Neo4j Desktop (database is local)
* Use the Neo4j Browser Web interface by specifying a URL in a Web browser using port 7474 (database is local, in Neo4j Aura, or in Neo4j Sandbox)

The user interface and behavior for the Neo4j Browser application and Neo4j Browser Web interface is identical, except that in the Web interface, you have the option of syncing your settings and scripts to the cloud.

### ****Guided Exercise: Getting Started with Neo4j Browser****

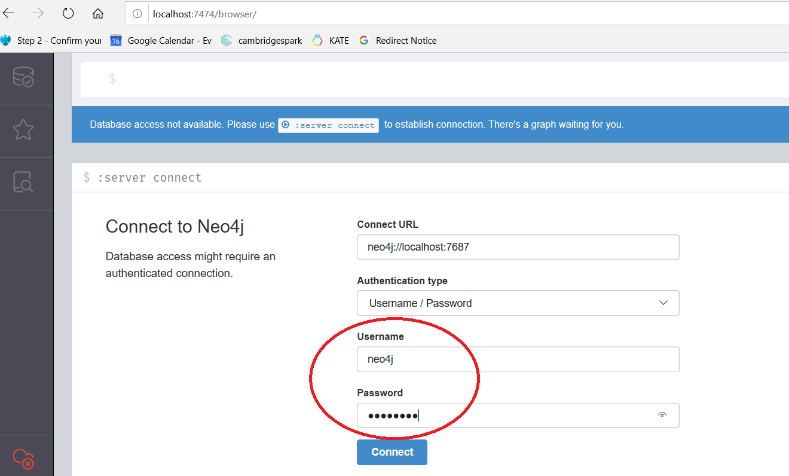
Follow along with this video to become familiar with common tasks in Neo4j Browser. You will use Neo4j Browser to populate the Movies database that is used for training as well as syncing your settings and scripts to the cloud.

Before you perform the tasks shown in this video, you must have either created and started the database in the Neo4j Desktop, created a Database in Neo4j Aura, or created a Neo4j Sandbox.

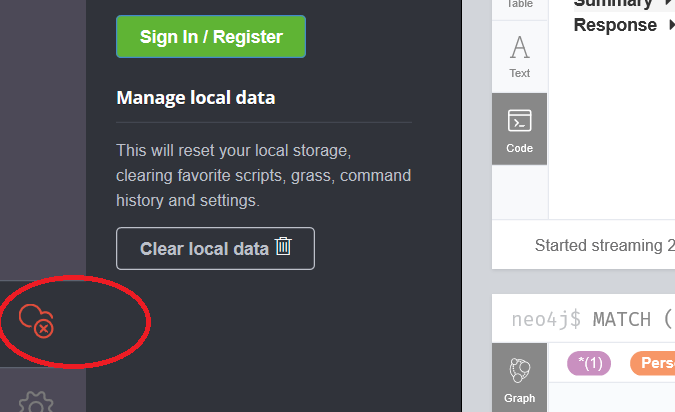
[ANA] To access the Neo4j Browser VIA WEB:

localhost:7474

username = neo4j and password = password (SET IN THE BROWSER!!)



[ANA] The only difference we see in the WEB browser is the cloud area:



[ANA] When we access the database for the first time the command below is executed:

:play start

A command begins with a colon => ‘:’

A Cypher statement does NOT begin with a colon!!!

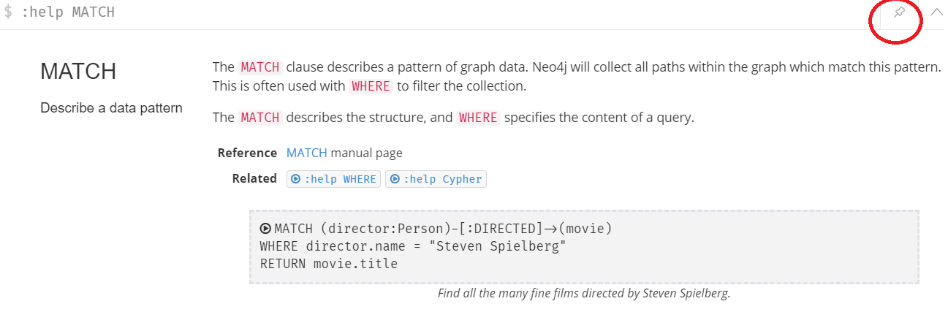
Try to type the commands below to get help for the command, keyboard shotcuts:

:help commands

:help keys

Quick help on Cypher keywords (MATCH = retrieves data from the database):

:help MATCH



The pane can be “pinned” it will always stay at this position. The result of the next command executed is placed below the pinned pane. SYSINFO shows info about the databases!

:sysinfo



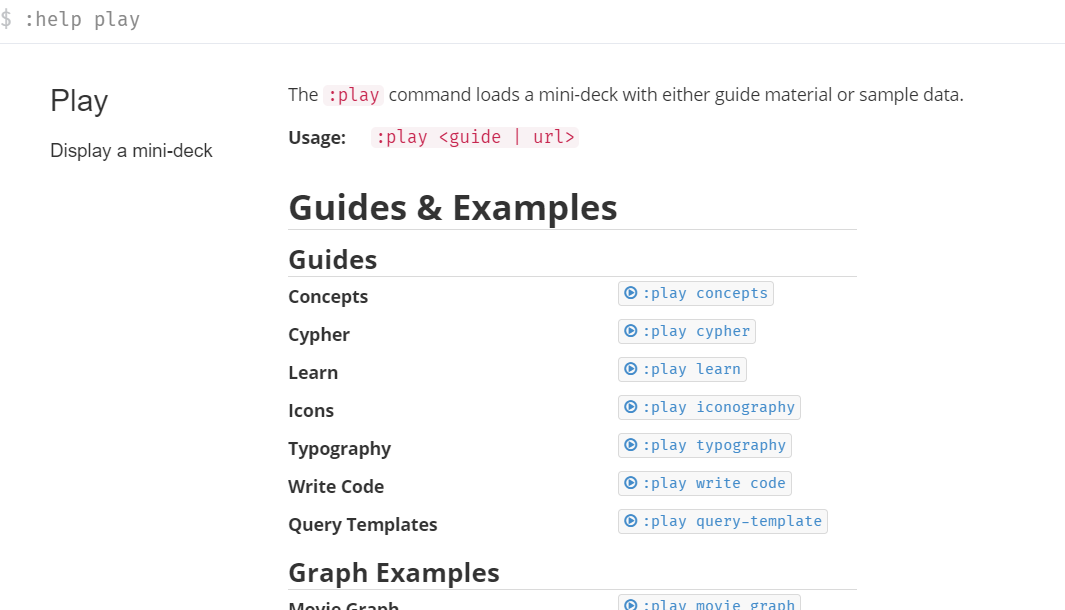
The command :history brings all commands, then just click on the one you want to run again:

:history

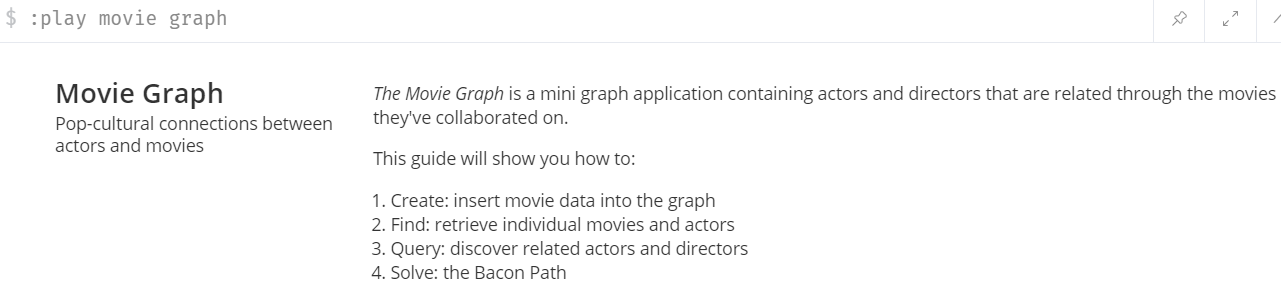
In SETTINGS we can change the # of result frames, and other limits! The command below also clears all the frames:

:clear

The :play command will execute a browser guide to show some help content



The command “:play movie” shows some things about CYPHER using the **movie** database



The command below does not return a graph, but a list of 5 “Persons”:

MATCH (p:Person) RETURN p.name LIMIT 5

To SAVE the commands I executed:

1. Click the Favorites icon on the left t
2. Click in the favorites TAB, create a folder to save my SCRIPT called Movies
3. RECALL the command that I used to create the movie DB on the right pane
4. To prevent another batch of the same records being re-created again, include the following line in the very beginning of the script:

MATCH (n) DETACH DELETE n;

1. (BEFORE that, go to SETTINGS and check the button “Enable multi statement query editor”)
2. Execute the script
3. Go back to the script and add the comment in the 1st line: “// reset database”
4. Then click the FAVORITE (star) icon
5. Then we can go to the favorites TAB and find the script there!!
6. Drag it and move it to the “Movies” folder

If you want to save your script in the CLOUD:

1. Click in the **cloud** button on the left pane
2. Sign in to the cloud
3. When you sign out all scripts will be saved in the cloud

MODULE 4

**Introduction to Cypher**

### About this module

Cypher is the query language you use to retrieve data from the Neo4j Database, as well as create and update the data.

At the end of this module, you should be able to write Cypher statements to:

* Retrieve nodes from the graph.
* Filter nodes retrieved using labels and property values of nodes.
* Retrieve property values from nodes in the graph.
* Filter nodes retrieved using relationships.

Throughout this training, you should refer to:

* [Neo4j Cypher Manual](https://neo4j.com/docs/cypher-manual/current/)
* [Cypher Reference card](http://neo4j.com/docs/cypher-refcard/current/)

### What is Cypher?

Cypher is a declarative query language that allows for expressive and efficient querying and updating of graph data. Cypher is a relatively simple and very powerful language. Complex database queries can easily be expressed through Cypher, allowing you to focus on your domain instead of getting lost in the syntax of database access.

Cypher is designed to be a human-friendly query language, suitable for both developers and other professionals. The guiding goal is to make the simple things easy, and the complex things possible.

#### **Cypher is ASCII art**

Optimized for being read by humans, Cypher’s construct uses English prose and iconography (called ASCII Art) to make queries more self-explanatory.

Being a declarative language, Cypher focuses on the clarity of expressing **what** to retrieve from a graph, not on **how** to retrieve it. You can think of Cypher as mapping English language sentence structure to patterns in a graph. For example, the nouns are nodes of the graph, the verbs are the relationships in the graph, and the adjectives and adverbs are the properties.

This is in contrast to imperative, programmatic APIs for database access. This approach makes query optimization an implementation detail instead of a burden on the developer, removing the requirement to update all traversals just because the physical database structure has changed.

Cypher is inspired by a number of different approaches and builds upon established practices for expressive querying. Many of the Cypher keywords like WHERE and ORDER BY are inspired by SQL. The pattern matching functionality of Cypher borrows concepts from SPARQL. And some of the collection semantics have been borrowed from languages such as Haskell and Python.

The Cypher language has been made available to anyone to implement and use via openCypher (opencypher.org), allowing any database vendor, researcher or other interested party to reap the benefits of our years of effort and experience in developing a first class graph query language.

#### **Nodes**

Cypher uses a pair of parentheses like (), (n) to represent a node, much like a circle on a whiteboard. Recall that a node typically represents an entity in your domain. An anonymous node, (), represents one or more nodes during a query processing where there are no restrictions of the type of node or the properties of the node. When you specify (n) for a node, you are telling the query processor that for this query, use the variable n to represent nodes that will be processed later in the query for further query processing or for returning values from the query.

#### **Labels**

Nodes in a graph are typically labeled. Labels are used to group nodes and filter queries against the graph. That is, labels can be used to optimize queries. In the Movie database you will be working with, the nodes in this graph are labeled Movie or Person to represent two types of nodes.

For example, you can see the labels in the database by simply clicking the Database icon in Neo4j Browser:

You can filter the types of nodes that you are querying, by specifying a **label** for a node. A node can have zero or more labels.

Here are simplified syntax examples for specifying a node:

()

(variable)

(:Label)

(variable:Label)

(:Label1:Label2)

(variable:Label1:Label2)

Notice that a node must have the parentheses. The labels and the variable for a node are optional.

Here are examples of specifying nodes in Cypher:

() // anonymous node not be referenced later in the query

(p) // variable p, a reference to a node used later

(:Person) // anonymous node of type Person

(p:Person) // p, a reference to a node of type Person

(p:Actor:Director) // p, a reference to a node of types Actor and Director

A node can have multiple labels. For example a node can be created with a label of Person and that same node can be modified to also have the label of Actor and/or Director.

#### **Comments in Cypher**

In Cypher, you can place a comment (starts with //) anywhere in your Cypher to specify that the rest of the line is interpreted as a comment.

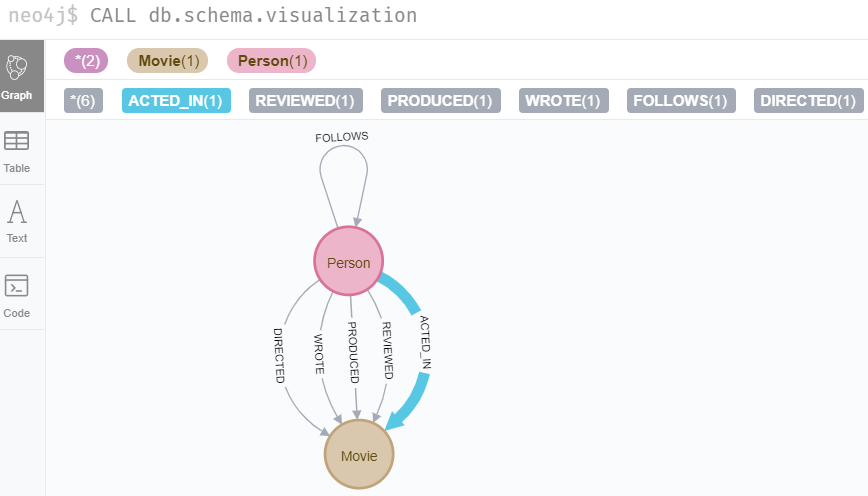
### Examining the data model

When you are first learning about the data (nodes, labels, etc.) in a graph, it is helpful to examine the data model of the graph. You do so by executing CALL db.schema, which calls the Neo4j procedure that returns information about the nodes, labels, and relationships in the graph.

For example, when we run this procedure in our training environment, we see the following in the result pane. Here we see that the graph has 2 labels defined for nodes, Person and Movie. Each type of nodes is displayed in a different color. The relationships between nodes are also displayed, which you will learn about later in this module.

[ANA] to get information about the DATA MODEL (not specific nodes):

CALL db.schema.visualization



### Using MATCH to retrieve nodes

In this video, you will be introduced to using the MATCH statement to retrieve nodes from the graph in Neo4j Browser.

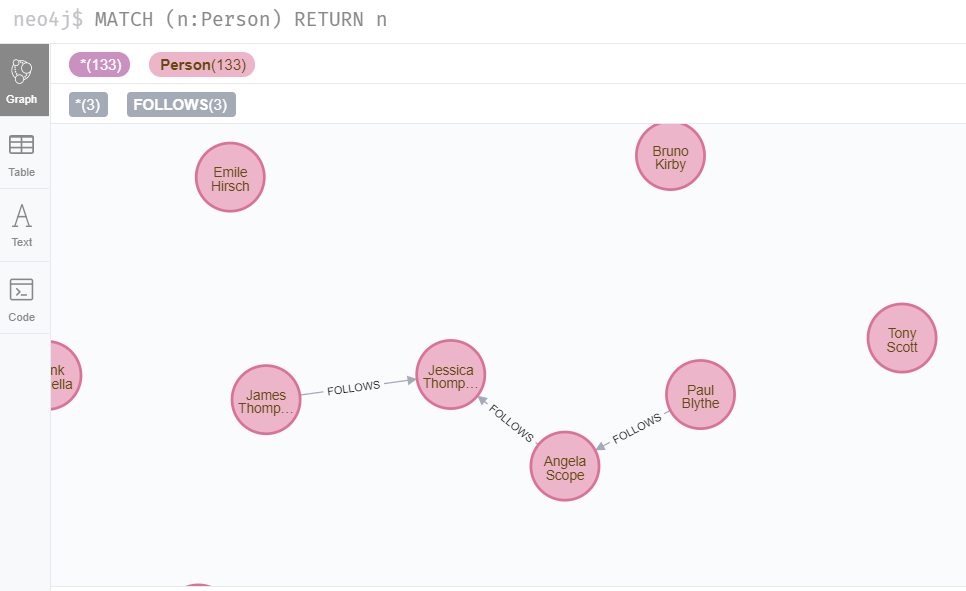
[ANA] When I click in the label “Person”, the following statement is automatically executed:

MATCH (n:Person) RETURN n LIMIT 25

It will also show relationships between the PERSON nodes (SETTING => “Connect result nodes” checkbox is checked!!!), there are THREE FOLLOWS relationships!

MATCH (n:Person) RETURN n

ZOOM out to find the relationships!!!



The most widely used Cypher clause is MATCH. The MATCH clause performs a pattern match against the data in the graph. During the query processing, the graph engine traverses the graph to find all nodes that match the graph pattern. As part of query, you can return nodes or data from the nodes using the RETURN clause. The RETURN clause must be the last clause of a query to the graph. Later in this training, you will learn how to use MATCH to select nodes and data for updating the graph. First, you will learn how to simply return nodes.

Here are simplified syntax examples for a query:

MATCH (variable)

RETURN variable

MATCH (variable:Label)

RETURN variable

Notice that the Cypher keywords MATCH and RETURN are upper-case. This coding convention is described in the Cypher Style Guide and will be used in this training. This MATCH clause returns all nodes in the graph, where the optional Label is used to return a subgraph if the graph contains nodes of different types. The variable must be specified here, otherwise the query will have nothing to return.

Here are example queries to the Movie database:

MATCH (n) // returns all nodes in the graph

RETURN n

MATCH (p:Person) // returns all Person nodes in the graph

RETURN p

When we execute the Cypher statement, MATCH (p:Person) RETURN p, the graph engine returns all nodes with the label Person. The default view of the returned nodes are the nodes that were referenced by the variable p.

When you specify a pattern for a MATCH clause, you should always specify a node label if possible. In doing so, the graph engine uses an index to retrieve the nodes which will perform better than not using a label for the MATCH.

#### **Viewing nodes as table data**

We can also view the nodes as table data where the nodes and their associated property values are shown in a JSON-style format:

When nodes are displayed as table values, the node labels and ids are not shown, only the property values for the nodes. Node ids are unique identifiers and are set by the graph engine when a node is created.

### ****Exercise 1: Retrieving nodes****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 1.

### Properties

In Neo4j, a node (and a relationship, which you will learn about later) can have properties that are used to further define a node. A property is identified by its property key. Recall that nodes are used to represent the entities of your business model. A property is defined for a node and not for a type of node. All nodes of the same type need not have the same properties.

For example, in the Movie graph, all Movie nodes have both title and released properties. However, it is not a requirement that every Movie node has a property, tagline.

Properties can be used to filter queries so that a subset of the graph is retrieved. In addition, with the RETURN clause, you can return property values from the retrieved nodes, rather than the nodes.

#### **Examining property keys**

As you prepare to create Cypher queries that use property values to filter a query, you can view the values for property keys of a graph by simply clicking the Database icon in Neo4j Browser. Alternatively, you can execute CALL db.propertyKeys, which calls the Neo4j library method that returns the property keys for the graph.

Here is what you will see in the result pane when you call the method to return the property keys in the Movie graph. This result stream contains all property keys in the graph. It does not display which nodes utilize these property keys.

#### **Retrieving nodes filtered by a property value**

You have learned previously that you can filter node retrieval by specifying a label. Another way you can filter a retrieval is to specify a value for a property. Any node that matches the value will be retrieved.

Here are simplified syntax examples for a query where we specify one or more values for properties that will be used to filter the query results and return a subset of the graph:

MATCH (variable {propertyKey: propertyValue})

RETURN variable

MATCH (variable:Label {propertyKey: propertyValue})

RETURN variable

MATCH (variable {propertyKey1: propertyValue1, propertyKey2: propertyValue2})

RETURN variable

MATCH (variable:Label {propertyKey: propertyValue, propertyKey2: propertyValue2})

RETURN variable

Here is an example where we filter the query results using a property value. We only retrieve Person nodes that have a born property value of 1970.

MATCH (p:Person {born: 1970})

RETURN p

Here is an example where we specify two property values for the query.

MATCH (m:Movie {released: 2003, tagline: 'Free your mind'})

RETURN m

As it turns out, there is only one movie with the tagline, Free your mind in the Movie database, but if we had specified a different year, the query would not have returned a value because when you specify properties, both properties must match.

#### **Returning property values**

In this video, you will see how to return property values to the output stream when you retrieve nodes from the graph in Neo4j Browser.

[ANA] retrieving specific properties (using ALIAS):

MATCH (m:Movie {released: 2006}) RETURN m.title, m.released AS `Year Released`

Thus far, you have seen how to retrieve nodes and return nodes (entire graph or a subset of the graph). You can use the RETURN clause to return property values of nodes retrieved.

Here are simplified syntax examples for returning property values, rather than nodes:

MATCH (variable {prop1: value})

RETURN variable.prop2

MATCH (variable:Label {prop1: value})

RETURN variable.prop2

MATCH (variable:Label {prop1: value, prop2: value})

RETURN variable.prop3

MATCH (variable {prop1:value})

RETURN variable.prop2, variable.prop3

In this example, we use the born property to refine the query, but rather than returning the nodes, we return the name and born values for every node that satisfies the query.

MATCH (p:Person {born: 1965})

RETURN p.name, p.born

#### **Specifying aliases**

If you want to customize the headings for a table containing property values, you can specify **aliases** for column headers.

Here is the simplified syntax for specifying an alias for a property value:

MATCH (variable:Label {propertyKey1: propertyValue1})

RETURN variable.propertyKey2 AS alias2

If you want a heading to contain a space between strings, you must specify the alias with the back tick ` character, rather than a single or double quote character. In fact, you can specify any variable, label, relationship type, or property key with a space also by using the back tick ` character.

Here we specify aliases for the returned property values:

MATCH (p:Person {born: 1965})

RETURN p.name AS name, p.born AS `birth year`

### ****Exercise 2: Filtering queries using property values****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 2.

### Relationships

Relationships are what make Neo4j graphs such a powerful tool for connecting complex and deep data. A relationship is a **directed** connection between two nodes that has a **relationship type** (name). In addition, a relationship can have properties, just like nodes. In a graph where you want to retrieve nodes, you can use relationships between nodes to filter a query.

#### **ASCII art**

Thus far, you have learned how to specify a node in a MATCH clause. You can specify nodes and their relationships to traverse the graph and quickly find the data of interest.

Here is how Cypher uses ASCII art to specify path used for a query:

() // a node

()--() // 2 nodes have some type of relationship

()-->() // the first node has a relationship to the second node

()<--() // the second node has a relationship to the first node

#### **Querying using relationships**

In your MATCH clause, you specify how you want a relationship to be used to perform the query. The relationship can be specified with or without direction.

Here are simplified syntax examples for retrieving a set of nodes that satisfy one or more directed and typed relationships:

MATCH (node1)-[:REL\_TYPE]->(node2)

RETURN node1, node2

MATCH (node1)-[:REL\_TYPEA | :REL\_TYPEB]->(node2)

RETURN node1, node2

where:

|  |  |
| --- | --- |
| *node1* | is a specification of a node where you may include node labels and property values for filtering. |
| *:REL\_TYPE* | is the type (name) for the relationship. For this syntax the relationship is from *node1* to *node2*. |
| *:REL\_TYPEA* , *:REL\_TYPEB* | are the relationships from *node1* to *node2.* The nodes are returned if at least one of the relationships exists. |
| *node2* | is a specification of a node where you may include node labels and property values for filtering. |

#### **Examining relationships**

You can run CALL db.schema to view the relationship types in the graph. In the Movie graph, we see these relationships between the nodes:

Here we see that this graph has a total of 6 relationship types between the nodes. Some Person nodes are connected to other Person nodes using the FOLLOWS relationship type. All of the other relationships in this graph are from Person nodes to Movie nodes.

#### **Using a relationship in a query**

Here is an example where we retrieve the Person nodes that have the ACTED\_IN relationship to the Movie, The Matrix. In other words, show me the actors that acted in The Matrix.

MATCH (p:Person)-[rel:ACTED\_IN]->(m:Movie {title: 'The Matrix'})

RETURN p, rel, m

For this query, we are using the variable p to represent the Person nodes during the query, the variable m to represent the Movie node retrieved, and the variable rel to represent the relationship for the relationship type, ACTED\_IN. We return a graph with the Person nodes, the Movie node and their ACTED\_IN relationships.

**Important:** You specify node labels whenever possible in your queries as it optimizes the retrieval in the graph engine. That is, you should **not** specify this same query as:

MATCH (p)-[rel:ACTED\_IN]->(m {title:'The Matrix'})

RETURN p,m

#### **Querying by multiple relationships**

Here is another example where we want to know the movies that Tom Hanks acted in and directed:

MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED\_IN|:DIRECTED]->(m:Movie)

RETURN p.name, m.title

Notice that there are multiple rows returned for the movie, That Thing You Do. This is because Tom Hanks acted in and directed that movie.

#### **Using anonymous nodes in a query**

Suppose you wanted to retrieve the actors that acted in The Matrix, but you do not need any information returned about the Movie node. You need not specify a variable for a node in a query if that node is not returned or used for later processing in the query. You can simply use the anonymous node in the query as follows:

MATCH (p:Person)-[:ACTED\_IN]->(:Movie {title: 'The Matrix'})

RETURN p.name

[ANA] ->(:Movie NO NODE VARIABLE SPECIfIED HERE!!!!

A best practice is to place named nodes (those with variables) before anonymous nodes in a MATCH clause.

#### **Using an anonymous relationship for a query**

Suppose you want to find all people who are in any way connected to the movie, The Matrix. You can specify an empty relationship type in the query so that all relationships are traversed and the appropriate results are returned. In this example, we want to retrieve all Person nodes that have any type of connection to the Movie node, with the title, The Matrix. This query returns more nodes with the relationships types, DIRECTED, ACTED\_IN, and PRODUCED.

MATCH (p:Person)-->(m:Movie {title: 'The Matrix'})

RETURN p, m

Here are other examples of using the anonymous relationship:

MATCH (p:Person)--(m:Movie {title: 'The Matrix'})

RETURN p, m

MATCH (m:Movie)<--(p:Person {name: 'Keanu Reeves'})

RETURN p, m

In this training, we will use -->, --, and <-- to represent anonymous relationships as it is a Cypher best practice.

#### **Retrieving the relationship types**

There is a built-in function, type() that returns the relationship type of a relationship.

Here is an example where we use the rel variable to hold the relationships retrieved. We then use this variable to return the relationship types.

MATCH (p:Person)-[rel]->(:Movie {title:'The Matrix'})

RETURN p.name, type(rel)

#### **Retrieving properties for relationships**

Recall that a node can have as set of properties, each identified by its property key. Relationships can also have properties. This enables your graph model to provide more data about the relationships between the nodes.

Here is an example from the Movie graph. The movie, The Da Vinci Code has two people that reviewed it, Jessica Thompson and James Thompson. Each of these Person nodes has the REVIEWED relationship to the Movie node for The Da Vinci Code. Each relationship has properties that further describe the relationship using the summary and rating properties.

Just as you can specify property values for filtering nodes for a query, you can specify property values for a relationship. This query returns the name of of the person who gave the movie a rating of 65.

MATCH (p:Person)-[:REVIEWED {rating: 65}]->(:Movie {title: 'The Da Vinci Code'})

RETURN p.name

### Using patterns for queries

Thus far, you have learned how to specify nodes, properties, and relationships in your Cypher queries. Since relationships are directional, it is important to understand how patterns are used in graph traversal during query execution. How a graph is traversed for a query depends on what directions are defined for relationships and how the pattern is specified in the MATCH clause.

Here is an example of where the FOLLOWS relationship is used in the Movie graph. Notice that this relationship is directional.

We can perform a query that returns all Person nodes who follow Angela Scope:

MATCH (p:Person)-[:FOLLOWS]->(:Person {name:'Angela Scope'})

RETURN p

If we reverse the direction in the pattern, the query returns different results:

MATCH (p:Person)<-[:FOLLOWS]-(:Person {name:'Angela Scope'})

RETURN p

#### **Traversing relationships**

Since we have a graph, we can traverse through nodes to obtain relationships further into the traversal.

For example, we can write a Cypher query to return all followers of the followers of Jessica Thompson.

MATCH (p:Person)-[:FOLLOWS]->(:Person)-[:FOLLOWS]->(:Person {name:'Jessica Thompson'})

RETURN p

This query could also be modified to return each person along the path by specifying variables for the nodes and returning them. In addition, you can assign a variable to the path and return the path as follows:

MATCH path = (:Person)-[:FOLLOWS]->(:Person)-[:FOLLOWS]->(:Person {name:'Jessica Thompson'})

RETURN path

#### **Using relationship direction to optimize a query**

When querying the relationships in a graph, you can take advantage of the direction of the relationship to traverse the graph. For example, suppose we wanted to get a result stream containing rows of actors and the movies they acted in, along with the director of the particular movie.

Here is the Cypher query to do this. Notice that the direction of the traversal is used to focus on a particular movie during the query:

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)

RETURN a.name, m.title, d.name

### Cypher style recommendations

Here are the **Neo4j-recommended** Cypher coding standards that we use in this training:

* Node labels are CamelCase and begin with an upper-case letter (examples: Person, NetworkAddress). Note that node labels are case-sensitive.
* Property keys, variables, parameters, aliases, and functions are camelCase and begin with a lower-case letter (examples: businessAddress, title). Note that these elements are case-sensitive.
* Relationship types are in upper-case and can use the underscore. (examples: ACTED\_IN, FOLLOWS). Note that relationship types are case-sensitive and that you cannot use the “-” character in a relationship type.
* Cypher keywords are upper-case (examples: MATCH, RETURN). Note that Cypher keywords are case-insensitive, but a best practice is to use upper-case.
* String constants are in single quotes, unless the string contains a quote or apostrophe (examples: ‘The Matrix’, “Something’s Gotta Give”). Note that you can also escape single or double quotes within strings that are quoted with the same using a backslash character.
* Specify variables only when needed for use later in the Cypher statement.
* Place named nodes and relationships (that use variables) before anonymous nodes and relationships in your MATCH clauses when possible.
* Specify anonymous relationships with -->, --, or <--

Here is an example showing some best coding practices:

MATCH (:Person {name: 'Diane Keaton'})-[movRel:ACTED\_IN]->

(:Movie {title:"Something's Gotta Give"})

RETURN movRel.roles

We recommend that you follow the [Cypher Style Guide](https://github.com/opencypher/openCypher/blob/master/docs/style-guide.adoc) when writing your Cypher statements.

### ****Exercise 3: Filtering queries using relationships****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 3.

MODULE 5

# **Getting More Out of Queries**

### About this module

You have learned how to query nodes and relationships in a graph using simple patterns. You learned how to use node labels, relationship types, and properties to filter your queries. Cypher provides a rich set of MATCH clauses and keywords you can use to get more out of your queries.

At the end of this module, you should be able to write Cypher statements to:

* Filter queries using the WHERE clause
* Control query processing
* Control what results are returned
* Work with Cypher lists and dates

### Filtering queries using WHERE

You have learned how to specify values for properties of nodes and relationships to filter what data is returned from the MATCH and RETURN clauses. The format for filtering you have learned thus far only tests equality, where you must specify values for the properties to test with. What if you wanted more flexibility about how the query is filtered? For example, you want to retrieve all movies released after 2000, or retrieve all actors born after 1970 who acted in movies released before 1995. Most applications need more flexibility in how data is filtered.

The most common clause you use to filter queries is the WHERE clause that follows a MATCH clause. In the WHERE clause, you can place conditions that are evaluated at runtime to filter the query.

Previously, you learned to write simple query as follows:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie {released: 2008})

RETURN p, m

Here is one way you specify the same query using the WHERE clause:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.released = 2008

RETURN p, m

In this example, you can only refer to named nodes or relationships in a WHERE clause so remember that you must specify a variable for any node or relationship you are testing in the WHERE clause. The benefit of using a WHERE clause is that you can specify potentially complex conditions for the query.

For example:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.released = 2008 OR m.released = 2009

RETURN p, m

#### **Specifying ranges in WHERE clauses**

Not only can the equality = be tested, but you can test ranges, existence, strings, as well as specify logical operations during the query.

Here is an example of specifying a range for filtering the query:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.released >= 2003 AND m.released <= 2004

RETURN p.name, m.title, m.released

You can also specify the same query as:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE 2003 <= m.released <= 2004

RETURN p.name, m.title, m.released

You can specify conditions in a WHERE clause that return a value of true or false (for example predicates). For testing numeric values, you use the standard numeric comparison operators. Each condition can be combined for runtime evaluation using the boolean operators AND, OR, XOR, and NOT. There are a number of numeric functions you can use in your conditions. See the Neo4j Cypher Manual’s section Mathematical Functions for more information.

A special condition in a query is when the retrieval returns an unknown value called null. You should read the Neo4j Cypher Manual’s section Working with null to understand how null values are used at runtime.

#### **Testing labels**

Thus far, you have used the node labels to filter queries in a MATCH clause. You can filter node labels in the WHERE clause also:

For example, these two Cypher queries:

MATCH (p:Person)

RETURN p.name

MATCH (p:Person)-[:ACTED\_IN]->(:Movie {title: 'The Matrix'})

RETURN p.name

can be rewritten using WHERE clauses as follows:

MATCH (p)

WHERE p:Person

RETURN p.name

MATCH (p)-[:ACTED\_IN]->(m)

WHERE p:Person AND m:Movie AND m.title='The Matrix'

RETURN p.name

Not all node labels need to be tested during a query, but if your graph has multiple labels for the same node, filtering it by the node label will provide better query performance.

#### **Testing the existence of a property**

Recall that a property is associated with a particular node or relationship. A property is not associated with a node with a particular label or relationship type. In one of our queries earlier, we saw that the movie “Something’s Gotta Give” is the only movie in the Movie database that does not have a tagline property. Suppose we only want to return the movies that the actor, Jack Nicholson acted in with the condition that they must all have a tagline.

Here is the query to retrieve the specified movies where we test the existence of the tagline property:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name='Jack Nicholson' AND exists(m.tagline)

RETURN m.title, m.tagline

#### **Testing strings**

Cypher has a set of string-related keywords that you can use in your WHERE clauses to test string property values. You can specify STARTS WITH, ENDS WITH, and CONTAINS.

For example, to find all actors in the Movie database whose first name is Michael, you would write:

MATCH (p:Person)-[:ACTED\_IN]->()

WHERE p.name STARTS WITH 'Michael'

RETURN p.name

Note that the comparison of strings is case-sensitive. There are a number of string-related functions you can use to further test strings. For example, if you want to test a value, regardless of its case, you could call the toLower() function to convert the string to lower case before it is compared.

MATCH (p:Person)-[:ACTED\_IN]->()

WHERE toLower(p.name) STARTS WITH 'michael'

RETURN p.name

|  |  |
| --- | --- |
| NOTE | In this example where we are converting a property to lower case, if an index has been created for this property, it will not be used at runtime. |

See the *String functions* section of the *Neo4j Cypher Manual* for more information. It is sometimes useful to use the built-in string functions to modify the data that is returned in the query in the RETURN clause.

#### **Testing with regular expressions**

If you prefer, you can test property values using regular expressions. You use the syntax =~ to specify the regular expression you are testing with. Here is an example where we test the name of the Person using a regular expression to retrieve all Person nodes with a name property that begins with ‘Tom’:

MATCH (p:Person)

WHERE p.name =~'Tom.\*'

RETURN p.name

|  |  |
| --- | --- |
| NOTE | If you specify a regular expression. The index will never be used. In addition, the property value must fully match the regular expression. |

#### **Testing with patterns**

Sometimes during a query, you may want to perform additional filtering using the relationships between nodes being visited during the query. For example, during retrieval, you may want to exclude certain paths traversed. You can specify a NOT specifier on a pattern in a WHERE clause.

Here is an example where we want to return all Person nodes of people who wrote movies:

MATCH (p:Person)-[:WROTE]->(m:Movie)

RETURN p.name, m.title

Next, we modify this query to exclude people who directed that movie:

MATCH (p:Person)-[:WROTE]->(m:Movie)

WHERE NOT exists( (p)-[:DIRECTED]->(m) )

RETURN p.name, m.title

Here is another example where we want to find Gene Hackman and the movies that he acted in with another person who also directed the movie.

MATCH (gene:Person)-[:ACTED\_IN]->(m:Movie)<-[:ACTED\_IN]-(other:Person)

WHERE gene.name= 'Gene Hackman'

AND exists( (other)-[:DIRECTED]->(m) )

RETURN gene, other, m

#### **Testing with list values**

If you have a set of values you want to test with, you can place them in a list or you can test with an existing list in the graph.

You can define the list in the WHERE clause. During the query, the graph engine will compare each property with the values IN the list. You can place either numeric or string values in the list, but typically, elements of the list are of the same type of data. If you are testing with a property of a string type, then all the elements of the list should be strings.

In this example, we only want to retrieve Person nodes of people born in 1965 or 1970:

MATCH (p:Person)

WHERE p.born IN [1965, 1970]

RETURN p.name as name, p.born as yearBorn

You can also compare a value to an existing list in the graph.

We know that the :ACTED\_IN relationship has a property, roles that contains the list of roles an actor had in a particular movie they acted in. Here is the query we write to return the name of the actor who played Neo in the movie The Matrix:

MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE 'Neo' IN r.roles AND m.title='The Matrix'

RETURN p.name

|  |  |
| --- | --- |
| NOTE | There are a number of syntax elements of Cypher that we have not covered in this training. For example, you can specify CASE logic in your conditional testing for your WHERE clauses. You can learn more about these syntax elements in the *Neo4j Cypher Manual* and the *Cypher Refcard*. |

**Exercise 4: Filtering queries using the WHERE clause**

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 4.

**Retrieve all people who have produced a movie, but have not directed a movie, returning their names and the movie titles.**

**MATCH (a:Person)-[:PRODUCED]->(m:Movie)**

**WHERE NOT ((a)-[:DIRECTED]->(:Movie)) RETURN a.name, m.title**

**Retrieve the movies and their actors where one of the actors also directed the movie, returning the actors names, the director’s name, and the movie title.**

**MATCH (a1:Person)-[:ACTED\_IN]->(m:Movie)<-[:ACTED\_IN]-(a2:Person)**

**WHERE exists( (a2)-[:DIRECTED]->(m) )**

**RETURN a1.name as Actor, a2.name as `Actor/Director`, m.title as Movie**

**Retrieve the movies that have an actor’s role that is the name of the movie, return the movie title and the role.**

**MATCH (a:Person)-[r:ACTED\_IN]->(m:Movie)**

**WHERE m.title in r.roles**

**RETURN m.title as Movie, a.name as Actor**

Controlling query processing

Now that you have learned how to provide filters for your queries by testing properties, relationships, and patterns using the WHERE clause, you will learn some additional Cypher techniques for controlling what the graph engine does during the query.

#### **Specifying multiple MATCH patterns**

This MATCH clause includes a pattern specified by two paths separated by a comma:

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie),

(m:Movie)<-[:DIRECTED]-(d:Person)

WHERE m.released = 2000

RETURN a.name, m.title, d.name

If possible, you should write the same query as follows:

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)

WHERE m.released = 2000

RETURN a.name, m.title, d.name

There are, however, some queries where you will need to specify two or more patterns. Multiple patterns are used when a query is complex and cannot be satisfied with a single pattern. This is useful when you are looking for a specific node in the graph and want to connect it to a different node. You will learn about creating nodes and relationships later in this training.

#### **Example 1: Using two MATCH patterns**

Here are some examples of specifying two paths in a MATCH clause. In the first example, we want the actors that worked with Keanu Reeves to meet Hugo Weaving, who has worked with Keanu Reeves. Here we retrieve the actors who acted in the same movies as Keanu Reeves, but not when Hugo Weaving acted in the same movie. To do this, we specify two paths for the MATCH:

MATCH (keanu:Person)-[:ACTED\_IN]->(movie:Movie)<-[:ACTED\_IN]-(n:Person),

(hugo:Person)

WHERE keanu.name='Keanu Reeves' AND

hugo.name='Hugo Weaving'

AND NOT (hugo)-[:ACTED\_IN]->(movie)

RETURN n.name

When you perform this type of query, you may see a warning in the query edit pane stating that the pattern represents a cartesian product and may require a lot of resources to perform the query. You should only perform these types of queries if you know the data well and the implications of doing the query.

#### **Example 2: Using two MATCH patterns**

Here is another example where two patterns are necessary. Suppose we want to retrieve the movies that Meg Ryan acted in and their respective directors, as well as the other actors that acted in these movies. Here is the query to do this:

MATCH (meg:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(d:Person),

(other:Person)-[:ACTED\_IN]->(m)

WHERE meg.name = 'Meg Ryan'

RETURN m.title as movie, d.name AS director , other.name AS `co-actors`

#### **Setting path variables**

You have previously seen how you can assign a path used in a MATCH clause to a variable. This is useful if you want to reuse the path later in the same query or if you want to return the path. So the previous Cypher statement could return the path as follows:

MATCH megPath = (meg:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(d:Person),

(other:Person)-[:ACTED\_IN]->(m)

WHERE meg.name = 'Meg Ryan'

RETURN megPath

#### **Specifying varying length paths**

Any graph that represents social networking, trees, or hierarchies will most likely have multiple paths of varying lengths. Think of the connected relationship in LinkedIn and how connections are made by people connected to more people. The Movie database for this training does not have much depth of relationships, but it does have the :FOLLOWS relationship that you learned about earlier:

You write a MATCH clause where you want to find all of the followers of the followers of a *Person* by specifying a numeric value for the number of hops in the path. Here is an example where we want to retrieve all *Person* nodes that are exactly two hops away:

MATCH (follower:Person)-[:FOLLOWS\*2]->(p:Person)

WHERE follower.name = 'Paul Blythe'

RETURN p

If we had specified [:FOLLOWS\*] rather than [:FOLLOWS\*2], the query would return all *Person* nodes that are in the :FOLLOWS path from *Paul Blythe*.

Here are simplified syntax examples for how varying length patterns are specified in Cypher:

Retrieve all paths of any length with the relationship, *:RELTYPE* from *nodeA* to *nodeB* and beyond:

(nodeA)-[:RELTYPE\*]->(nodeB)

Retrieve all paths of any length with the relationship, *:RELTYPE* from *nodeA* to *nodeB* or from *nodeB* to *nodeA* and beyond. This is usually a very expensive query so you should place limits on how many nodes are retrieved:

(nodeA)-[:RELTYPE\*]-(nodeB)

Retrieve the paths of length 3 with the relationship, *:RELTYPE* from *nodeA* to *nodeB*:

(node1)-[:RELTYPE\*3]->(node2)

Retrieve the paths of lengths 1, 2, or 3 with the relationship, *:RELTYPE* from *nodeA* to *nodeB*, *nodeB* to *nodeC*, as well as, *nodeC* to \_nodeD) (up to three hops):

(node1)-[:RELTYPE\*1..3]->(node2)

You can learn more about varying paths in the *Patterns* section of the *Neo4j Cypher Manual*.

#### **Finding the shortest path**

A built-in function that you may find useful in a graph that has many ways of traversing the graph to get to the same node is the shortestPath() function. Using the shortest path between two nodes improves the performance of the query.

In this example, we want to discover a shortest path between the movies The Matrix and A Few Good Men. In our MATCH clause, we set the variable p to the result of calling shortestPath(), and then return p. In the call to shortestPath(), notice that we specify \* for the relationship. This means any relationship; for the traversal.

MATCH p = shortestPath((m1:Movie)-[\*]-(m2:Movie))

WHERE m1.title = 'A Few Good Men' AND

m2.title = 'The Matrix'

RETURN p

Notice that the graph engine has traversed many types of relationships to get to the end node.

When you use the shortestPath() function, the query editor will show a warning that this type of query could potentially run for a long time. You should heed the warning, especially for large graphs. Read the Graph Algorithms documentation about the shortest path algorithm.

When you use ShortestPath(), you can specify a upper limits for the shortest path. In addition, you should aim to provide the patterns for the from and to nodes that execute efficiently. For example, use labels and indexes.

#### **Specifying optional pattern matching**

OPTIONAL MATCH matches patterns with your graph, just like MATCH does. The difference is that if no matches are found, OPTIONAL MATCH will use NULLs for missing parts of the pattern. OPTIONAL MATCH could be considered the Cypher equivalent of the outer join in SQL.

Here is an example where we query the graph for all people whose name starts with James. The OPTIONAL MATCH is specified to include people who have reviewed movies:

MATCH (p:Person)

WHERE p.name STARTS WITH 'James'

OPTIONAL MATCH (p)-[r:REVIEWED]->(m:Movie)

RETURN p.name, type(r), m.title

Notice that for all rows that do not have the :REVIEWED relationship, a null value is returned for the movie part of the query, as well as the relationship.

#### **Aggregation in Cypher**

Aggregation in Cypher is different from aggregation in SQL. In Cypher, you need not specify a grouping key. As soon as an aggregation function is used, all non-aggregated result columns become grouping keys. The grouping is implicitly done, based upon the fields in the RETURN clause.

For example, in this Cypher statement, all rows returned with the same values for a.name and d.name are counted and only returned once.

// implicitly groups by a.name and d.name

MATCH (a)-[:ACTED\_IN]->(m)<-[:DIRECTED]-(d)

RETURN a.name, d.name, count(\*)

#### **Collecting results**

Cypher has a built-in function, collect() that enables you to aggregate a value into a list. Here is an example where we collect the list of movies that Tom Cruise acted in:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name ='Tom Cruise'

RETURN collect(m.title) AS `movies for Tom Cruise`

In Cypher, there is no “GROUP BY” clause as there is in SQL. The graph engine uses non-aggregated columns as an automatic grouping key.

#### **Counting results**

The Cypher count() function is very useful when you want to count the number of occurrences of a particular query result. If you specify count(n), the graph engine calculates the number of occurrences of n. If you specify count(\*), the graph engine calculates the number of rows retrieved, including those with null values. When you use count(), the graph engine does an implicit group by based upon the aggregation.

Here is an example where we count the paths retrieved where an actor and director collaborated in a movie and the count() function is used to count the number of paths found for each actor/director collaboration.

MATCH (actor:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(director:Person)

RETURN actor.name, director.name, count(m) AS collaborations, collect(m.title) AS movies

There are more aggregating functions such as min() or max() that you can also use in your queries. These are described in the Aggregating Functions section of the Neo4j Cypher Manual.

#### **Additional processing using WITH**

During the execution of a MATCH clause, you can specify that you want some intermediate calculations or values that will be used for further processing of the query, or for limiting the number of results before further processing is done. You use the WITH clause to perform intermediate processing or data flow operations.

Here is an example where we start the query processing by retrieving all actors and their movies. During the query processing, want to only return actors that have 2 or 3 movies. All other actors and the aggregated results are filtered out. This type of query is a replacement for SQL’s “HAVING” clause. The WITH clause does the counting and collecting, but is then used in the subsequent WHERE clause to limit how many paths are visited.

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WITH a, count(a) AS numMovies, collect(m.title) as movies

WHERE numMovies > 1 AND numMovies < 4

RETURN a.name, numMovies, movies

When you use the WITH clause, you specify the variables from the previous part of the query you want to pass on to the next part of the query. In this example, the variable *a* is specified to be passed on in the query, but *m* is not. Since *m* is not specified to be passed on, *m* will not be available later in the query. Notice that for the RETURN clause, *a*, *numMovies*, and *movies* are available for use.

|  |  |
| --- | --- |
| NOTE | You have to name all expressions with an alias in a WITH that are not simple variables. |

Here is another example where we want to find all actors who have acted in at least five movies, and find (optionally) the movies they directed and return the person and those movies.

MATCH (p:Person)

WITH p, size((p)-[:ACTED\_IN]->(:Movie)) AS movies

WHERE movies >= 5

OPTIONAL MATCH (p)-[:DIRECTED]->(m:Movie)

RETURN p.name, m.title

In this example, we first retrieve all people, but then specify a pattern in the WITH clause where we calculate the number of :ACTED\_IN relationships retrieved using the size() function. If this value is greater than five, we then also retrieve the :DIRECTED paths to return the name of the person and the title of the movie they directed. In the result, we see that these actors acted in more than five movies, but Tom Hanks is the only actor who directed a movie and thus the only person to have a value for the movie.

### ****Exercise 5: Controlling query processing****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 5.

1)

MATCH (gene:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(d:Person),

(other:Person)-[:ACTED\_IN]->(m)

WHERE gene.name = 'Gene Hackman'

RETURN m.title, d.name AS director, other.name as `CO-ACTORS`

2)

MATCH (p1:Person)-[:FOLLOWS]-(p2:Person)

WHERE p1.name = 'James Thompson'

RETURN p1, p2

3)

MATCH (p1:Person)-[:FOLLOWS\*3]-(p2:Person)

WHERE p1.name = 'James Thompson'

RETURN p1, p2

4)

MATCH (p1:Person)-[:FOLLOWS\*1..2]-(p2:Person)

WHERE p1.name = 'James Thompson'

RETURN p1, p2

5)

MATCH (p1:Person)-[:FOLLOWS\*]-(p2:Person)

WHERE p1.name = 'James Thompson'

RETURN p1, p2

6)

MATCH (p:Person)

WHERE p.name STARTS WITH 'Tom'

OPTIONAL MATCH (p:Person)-[:DIRECTED]->(m:Movie)

RETURN p.name, m.title

7)

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

RETURN p.name, collect(m.title) AS `movies acted`

8)

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)<-[:ACTED\_IN]-(p2:Person)

WHERE p.name = 'Tom Cruise'

RETURN m.title, collect(p2.name) AS `co-actors`

9)

MATCH (a:Person)-[:REVIEWED]->(m:Movie)

WITH m, collect(a.name) as reviewers, count(a) AS numReviews

RETURN m.title, reviewers, numReviews

10)

MATCH (d:Person)-[:DIRECTED]->(m:Movie)<-[:ACTED\_IN]-(a:Person)

WITH d, collect(a.name) as actors, count(a) AS numActors

RETURN d.name AS director, numActors, actors

11)

MATCH (d:Person)-[:ACTED\_IN]->(m:Movie)

WITH d, collect(m.title) as listMovies, count(m) AS numMovies

WHERE numMovies = 5

RETURN d.name AS actor, listMovies

12)

MATCH (m:Movie)

WITH m, size((:Person)-[:DIRECTED]->(m)) AS directors

WHERE directors >= 2

OPTIONAL MATCH (p:Person)-[:REVIEWED]->(m)

RETURN m.title, p.name

### Controlling how results are returned

Next, you will learn some additional Cypher techniques for controlling how results are returned from a query.

#### **Eliminating duplication**

You have seen a number of query results where there is duplication in the results returned. In most cases, you want to eliminate duplicated results. You do so by using the DISTINCT keyword.

Here is a simple example where duplicate data is returned. Tom Hanks both acted in and directed the movie, That Thing You Do, so the movie is returned twice in the result stream:

MATCH (p:Person)-[:DIRECTED | :ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

RETURN m.released, collect(m.title) AS movies

We can eliminate the duplication by specifying the DISTINCT keyword as follows:

MATCH (p:Person)-[:DIRECTED | :ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

RETURN m.released, collect(DISTINCT m.title) AS movies

#### **Using WITH and DISTINCT to eliminate duplication**

Another way that you can avoid duplication is to with WITH and DISTINCT together as follows:

MATCH (p:Person)-[:DIRECTED | :ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

WITH DISTINCT m

RETURN m.released, m.title

#### **Ordering results**

If you want the results to be sorted, you specify the expression to use for the sort using the ORDER BY keyword and whether you want the order to be descending using the DESC keyword. Ascending order is the default. Note that you can provide multiple sort expressions and the result will be sorted in that order. Just as you can use DISTINCT with WITH to eliminate duplication, you can use ORDER BY with WITH to control the sorting of results.

In this example, we specify that the release date of the movies for Tom Hanks will be returned in descending order.

MATCH (p:Person)-[:DIRECTED | :ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

RETURN m.released, collect(DISTINCT m.title) AS movies ORDER BY m.released DESC

#### **Limiting the number of results**

Although you can filter queries to reduce the number of results returned, you may also want to limit the number of results. This is useful if you have very large result sets and you only need to see the beginning or end of a set of ordered results. You can use the LIMIT keyword to specify the number of results returned. Furthermore, you can use the LIMIT keyword with the WITH clause to limit results.

Suppose you want to see the titles of the ten most recently released movies. You could do so as follows where you limit the number of results using the LIMIT keyword as follows:

MATCH (m:Movie)

RETURN m.title as title, m.released as year ORDER BY m.released DESC LIMIT 10

#### **Controlling the number of results using WITH**

Previously, you saw how you can use the WITH clause to perform some intermediate processing during a query. You can use the WITH clause to limit the number of results.

In this example, we count the number of movies during the query and we return the results once we have reached 5 movies:

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WITH a, count(\*) AS numMovies, collect(m.title) as movies

WHERE numMovies = 5

RETURN a.name, numMovies, movies

### ****Exercise 6: Controlling results returned****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 6.

1)

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.released >= 1990 AND m.released < 2000

RETURN m.released, m.title, collect(a.name)

2)

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.released >= 1990 AND m.released < 2000

RETURN m.released, collect(m.title), collect(a.name)

3)

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.released >= 1990 AND m.released < 2000

RETURN m.released, collect(DISTINCT m.title), collect(a.name)

4)

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.released >= 1990 AND m.released < 2000

RETURN m.released, collect(DISTINCT m.title), collect(a.name)

ORDER BY m.released DESC

5)

MATCH (p:Person)-[r:REVIEWED ]->(m:Movie)

RETURN r.rating, m.title

ORDER BY r.rating DESC

LIMIT 5

6)

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WITH a, count(a) AS numMovies, collect(m.title) AS movies

WHERE numMovies <= 3

RETURN a.name, movies

### Working with Cypher data

Thus far, you have specified both string and numeric types in your Cypher queries. You have also learned that nodes and relationships can have properties, whose values are structured like JSON objects. You have also learned that the collect() function can create lists of values or objects where a list is comma-separated and you can use the IN keyword to search for a value in a list. Next, you will learn more about working with lists and dates in Cypher.

#### **Lists**

There are many built-in Cypher functions that you can use to build or access elements in lists. A Cypher map is list of key/value pairs where each element of the list is of the format key: value. For example, a map of months and the number of days per month could be:

[Jan: 31, Feb: 28, Mar: 31, Apr: 30 , May: 31, Jun: 30 , Jul: 31, Aug: 31, Sep: 30, Oct: 31, Nov: 30, Dec: 31]

You can collect values for a list during a query and when you return results, you can sort by the size of the list using the size() function as follows:

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WITH m, count(m) AS numCast, collect(a.name) as cast

RETURN m.title, cast, numCast ORDER BY size(cast)

You can read more about working with lists in the List Functions section of the Neo4j Cypher Manual.

#### **Unwinding lists**

There may be some situations where you want to perform the opposite of collecting results, but rather separate the lists into separate rows. This functionality is done using the UNWIND clause.

Here is an example where we create a list with three elements, unwind the list and then return the values. Since there are three elements, three rows are returned with the values:

WITH [1, 2, 3] AS list

UNWIND list AS row

RETURN list, row

Notice that there is no MATCH clause. You need not query the database to execute Cypher statements, but you do need the RETURN clause here to return the calculated values from the Cypher query.

|  |  |
| --- | --- |
| NOTE | The UNWIND clause is frequently used when importing data into a graph. |

#### **Dates**

Cypher has a built-in date() function, as well as other temporal values and functions that you can use to calculate temporal values. You use a combination of numeric, temporal, spatial, list and string functions to calculate values that are useful to your application. For example, suppose you wanted to calculate the age of a Person node, given a year they were born (the born property must exist and have a value).

Here is example Cypher to retrieve all actors from the graph, and if they have a value for born, calculate the age value.

MATCH (actor:Person)-[:ACTED\_IN]->(:Movie)

WHERE exists(actor.born)

// calculate the age

with DISTINCT actor, date().year - actor.born as age

RETURN actor.name, age as `age today`

ORDER BY actor.born DESC

Consult the *Neo4j Cypher Manual* for more information about the built-in functions available for working with data of all types:

* Predicate
* Scalar
* List
* Mathematical
* String
* Temporal
* Spatial

**Exercise 7: Working with Cypher data**

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 7.

1)

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie),

(m)<-[:PRODUCED]-(p:Person)

WITH m, collect(DISTINCT a.name) AS cast, collect(DISTINCT p.name) AS producers

RETURN DISTINCT m.title, cast, producers

ORDER BY size(cast)

2)

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WITH p, collect(m) AS movies

WHERE size(movies) > 5

RETURN p.name, movies

3)

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WITH p, collect(m) AS movies

WHERE size(movies) > 5

WITH p, movies UNWIND movies as movie

RETURN p.name, movie.title

4)

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WHERE a.name = 'Tom Hanks'

RETURN m.title, m.released, date().year - m.released as yearsAgoReleased, m.released - a.born AS `age of Tom`

ORDER BY yearsAgoReleased

MODULE 6

# **Creating Nodes and Relationships**

### About this module

You have learned how to query a graph using a number of Cypher clauses and keywords that start with the MATCH clause. You learned how to retrieve data based upon label values, property key values, and relationship types, and how to perform some useful intermediate processing during a query to control what data is returned.

At the end of this module, you should be able to write Cypher statements to:

* Create a node
  + Add and remove node labels
  + Add and remove node properties
  + Update properties
* Create a relationship
  + Add and remove properties for a relationship
* Delete a node
* Delete a relationship
* Merge data in a graph
  + Creating nodes
  + Creating relationships

### Creating nodes

Recall that a node is an element of a graph representing a domain entity that has zero or more labels, properties, and relationships to or from other nodes in the graph.

When you create a node, you can add it to the graph without connecting it to another node.

Here is the simplified syntax for creating a node:

CREATE (optionalVariable optionalLabels {optionalProperties})

If you plan on referencing the newly created node, you must provide a variable. Whether you provide labels or properties at node creation time is optional. In most cases, you will want to provide some label and property values for a node when created. This will enable you to later retrieve the node. Provided you have a reference to the node (for example, using a MATCH clause), you can always add, update, or remove labels and properties at a later time.

Here are some examples of creating a single node in Cypher:

Add a node to the graph of type Movie with the title Batman Begins. This node can be retrieved using the title. A set of nodes with the label Movie can also be retrieved which will contain this node:

CREATE (:Movie {title: 'Batman Begins'})

Add a node with two labels to the graph of types *Movie* and *Action* with the *title* *Batman Begins*. This node can be retrieved using the title. A set of nodes with the labels *Movie* or *Action* can also be retrieved which will contain this node:

CREATE (:Movie:Action {title: 'Batman Begins'})

Add a node to the graph of types *Movie* and *Action* with the *title* *Batman Begins*. This node can be retrieved using the title. A set of nodes with the labels *Movie* or *Action* can also be retrieved which will contain this node. The variable *m* can be used for later processing after the CREATE clause:

CREATE (m:Movie:Action {title: 'Batman Begins'})

Add a node to the graph of types *Movie* and *Action* with the *title* *Batman Begins*. This node can be retrieved using the title. A set of nodes with the labels *Movie* or *Action* can also be retrieved which will contain this node. Here we return the title of the node:

CREATE (m:Movie:Action {title: ' Batman Begins'})

RETURN m.title

When the graph engine creates a node, it automatically assigns a read-only, unique ID to the node. Here we see that the id of the node is 568. This is not a property of a node, but rather an internal value.

After you have created a node, you can add more properties or labels to it and most importantly, connect it to another node.

### Creating multiple nodes

You can create multiple nodes by simply separating the nodes specified with commas, or by specifying multiple CREATE statements.

Here is an example, where we create some Person nodes that will represent some of the people associated with the movie Batman Begins:

CREATE

(:Person {name: 'Michael Caine', born: 1933}),

(:Person {name: 'Liam Neeson', born: 1952}),

(:Person {name: 'Katie Holmes', born: 1978}),

(:Person {name: 'Benjamin Melniker', born: 1913})

|  |
| --- |
| The graph engine will create a node with the same properties of a node that already exists. You can prevent this from happening in one of two ways:   1. You can use MERGE rather than CREATE when creating the node.   2. You can add constraints to your graph. |

You will learn about merging data later in this module. Constraints are configured globally for a graph and are covered later in this training.

### Adding labels to a node

You may not know ahead of time what label or labels you want for a node when it is created. You can add labels to a node using the SET clause.

Here is the simplified syntax for adding labels to a node:

SET x:Label // adding one label to node referenced by the variable x

SET x:Label1:Label2 // adding two labels to node referenced by the variable x

If you attempt to add a label to a node for which the label already exists, the SET processing is ignored.

Here is an example where we add the *Action* label to the node that has a label, *Movie*:

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m:Action

RETURN labels(m)

Notice here that we call the built-in function, labels() that returns the set of labels for the node.

### Removing labels from a node

Perhaps your data model has changed or the underlying data for a node has changed so that the label for a node is no longer useful or valid.

Here is the simplified syntax for removing labels from a node:

REMOVE x:Label // remove the label from the node referenced by the variable x

If you attempt to remove a label from a node for which the label does not exist, the SET processing is ignored.

Here is an example where we remove the Action label from the node that has a labels, Movie and Action:

MATCH (m:Movie:Action)

WHERE m.title = 'Batman Begins'

REMOVE m:Action

RETURN labels(m)

### Adding properties to a node

After you have created a node and have a reference to the node, you can add properties to the node, again using the SET keyword.

Here are simplified syntax examples for adding properties to a node referenced by the variable x:

SET x.propertyName = value

SET x.propertyName1 = value1 , x.propertyName2 = value2

SET x = {propertyName1: value1, propertyName2: value2}

SET x += {propertyName1: value1, propertyName2: value2}

If the property does not exist, it is added to the node. If the property exists, its value is updated. If the value specified is null, the property is removed.

Note that the type of data for a property is not enforced. That is, you can assign a string value to a property that was once a numeric value and visa versa.

When you specify the JSON-style object for assignment (using =) of the property values for the node, the object must include all of the properties and their values for the node as the existing properties for the node are overwritten. However, if you specify += when assigning to a property, the value at valueX is updated if the propertyNnameX exists for the node. If the propertyNameX does not exist for the node, then the property is added to the node.

Here is an example where we add the properties released and lengthInMinutes to the movie Batman Begins:

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m.released = 2005, m.lengthInMinutes = 140

RETURN m

Here is another example where we set the property values to the movie node using the JSON-style object containing the property keys and values. Note that all properties must be included in the object.

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m = {title: 'Batman Begins',

released: 2005,

lengthInMinutes: 140,

videoFormat: 'DVD',

grossMillions: 206.5}

RETURN m

Note that when you add a property to a node for the first time in the graph, the property key is added to the graph. So for example, in the previous example, we added the videoFormat and grossMillions property keys to the graph as they have never been used before for a node in the graph. Once a property key is added to the graph, it is never removed. When you examine the property keys in the database (by executing CALL db.propertyKeys(), you will see all property keys created for the graph, regardless of whether they are currently used for nodes and relationships.

Here is an example where we use the JSON-style object to add the *awards* property to the node and update the *grossMillions* property:

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m += { grossMillions: 300,

awards: 66}

RETURN m

### Removing properties from a node

There are two ways that you can remove a property from a node. One way is to use the REMOVE keyword. The other way is to set the property’s value to null.

Here are simplified syntax examples for removing properties from a node referenced by the variable x:

REMOVE x.propertyName

SET x.propertyName = null

Suppose we determined that no other Movie node in the graph has the properties, videoFormat and grossMillions. There is no restriction that nodes of the same type must have the same properties. However, we have decided that we want to remove these properties from this node. Here is example Cypher to remove this property from this Batman Begins node:

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m.grossMillions = null

REMOVE m.videoFormat

RETURN m

Assuming that we have previously created the node for the movie with the these properties, here is the result of running this Cypher statement where we remove each property a different way. One way we remove the property using the SET clause to set the property to null. And in another way, we use the REMOVE clause.

### ****Exercise 8: Creating Nodes****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 8.

1)

CREATE (m:Movie {title: 'Forrest Gump'})

RETURN m

2)

MATCH (m:Movie)

WHERE m.title = 'Forrest Gump'

RETURN m

3)

CREATE (p:Person {name:'Robin Wright'})

RETURN p

4)

MATCH (p:Person) WHERE p.name = 'Robin Wright'

RETURN p

5)

MATCH (m:Movie)

WHERE m.released < 2010

SET m:OlderMovie

RETURN DISTINCT labels(m)

6)

MATCH (m:OlderMovie)

RETURN m.title, m.released

7)

MATCH (p:Person)

WHERE p.name STARTS WITH 'Robin'

SET p:Female

RETURN DISTINCT labels(p)

8)

MATCH (p:Female)

RETURN p.name

9)

MATCH (p:Female)

REMOVE p:Female

10)

CALL db.schema.visualization

11)

MATCH (m:Movie)

WHERE m.title = 'Forrest Gump'

SET m:OlderMovie,

m.released = 1994,

m.tagline = "Life is like a box of chocolates...you never know what you're gonna get.",

m.lengthInMinutes = 142

12)

MATCH (m:OlderMovie)

WHERE m.title = 'Forrest Gump'

RETURN m

13)

MATCH (p:Person)

WHERE p.name = 'Robin Wright'

SET p.born = 1966, p.birthPlace = 'Dallas'

14)

MATCH (p:Person)

WHERE p.name = 'Robin Wright'

RETURN p

15)

MATCH (m:Movie)

WHERE m.title = 'Forrest Gump'

SET m.lengthInMinutes = null

16)

MATCH (m:Movie)

WHERE m.title = 'Forrest Gump'

RETURN m

17)

ATCH (p:Person)

WHERE p.name = 'Robin Wright'

REMOVE p.birthPlace

18)

MATCH (p:Person)

WHERE p.name = 'Robin Wright'

RETURN p

### Creating relationships

As you have learned in the previous exercises where you query the graph, you often query using connections between nodes. The connections capture the semantic relationships and context of the nodes in the graph.

Here is the simplified syntax for creating a relationship between two nodes referenced by the variables x and y:

CREATE (x)-[:REL\_TYPE]->(y)

CREATE (x)<-[:REL\_TYPE]-(y)

When you create the relationship, it must have direction. You can query nodes for a relationship in either direction, but you must create the relationship with a direction. An exception to this is when you create a node using MERGE that you will learn about later in this module.

In most cases, unless you are connecting nodes at creation time, you will retrieve the two nodes, each with their own variables, for example, by specifying a WHERE clause to find them, and then use the variables to connect them.

Here is an example. We want to connect the actor, *Michael Caine* with the movie, *Batman Begins*. We first retrieve the nodes of interest, then we create the relationship:

MATCH (a:Person), (m:Movie)

WHERE a.name = 'Michael Caine' AND m.title = 'Batman Begins'

CREATE (a)-[:ACTED\_IN]->(m)

RETURN a, m

|  |
| --- |
| Before you run these Cypher statements, you may see a warning in Neo4j Browser that you are creating a query that is a cartesian product that could potentially be a performance issue. You will see this warning if you have no unique constraint on the lookup keys. You will learn about uniqueness constraints later in the next module. If you are familiar with the data in the graph and can be sure that the MATCH clauses will not retrieve large amounts of data, you can continue. In our case, we are simply looking up a particular *Person* node and a particular *Movie* node so we can create the relationship. |

You can create multiple relationships at once by simply providing the pattern for the creation that includes the relationship types, their directions, and the nodes that you want to connect.

Here is an example where we have already created *Person* nodes for an actor, *Liam Neeson*, and a producer, *Benjamin Melniker*. We create two relationships in this example, one for *ACTED\_IN* and one for *PRODUCED*.

MATCH (a:Person), (m:Movie), (p:Person)

WHERE a.name = 'Liam Neeson' AND

m.title = 'Batman Begins' AND

p.name = 'Benjamin Melniker'

CREATE (a)-[:ACTED\_IN]->(m)<-[:PRODUCED]-(p)

RETURN a, m, p

|  |  |
| --- | --- |
|  | When you create relationships based upon a MATCH clause, you must be certain that only a single node is returned for the MATCH, otherwise multiple relationships will be created. |

### Adding properties to relationships

You can add properties to a relationship, just as you add properties to a node. You use the SET clause to do so.

Here is the simplified syntax for adding properties to a relationship referenced by the variable r:

SET r.propertyName = value

SET r.propertyName1 = value1 , r.propertyName2 = value2

SET r = {propertyName1: value1, propertyName2: value2}

SET r += {propertyName1: value1, propertyName2: value2}

If the property does not exist, it is added to the relationship. If the property exists, its value is updated for the relationship. When you specify the JSON-style object for assignment to the relationship using =, the object must include all of the properties for the relationship, just as you need to do for nodes. If you use +=, you can add or update properties, just as you do for nodes.

Here is an example where we will add the *roles* property to the *ACTED\_IN* relationship from *Christian Bale* to *Batman Begins* right after we have created the relationship:

MATCH (a:Person), (m:Movie)

WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'

CREATE (a)-[rel:ACTED\_IN]->(m)

SET rel.roles = ['Bruce Wayne','Batman']

RETURN a, m

The *roles* property is a list so we add it as such. If the relationship had multiple properties, we could have added them as a comma separated list or as an object, like you can do for node properties.

You can also add properties to a relationship when the relationship is created. Here is another way to create and add the properties for the relationship:

MATCH (a:Person), (m:Movie)

WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'

CREATE (a)-[:ACTED\_IN {roles: ['Bruce Wayne', 'Batman']}]->(m)

RETURN a, m

By default, the graph engine will create a relationship between two nodes, even if one already exists. You can test to see if the relationship exists before you create it as follows:

MATCH (a:Person),(m:Movie)

WHERE a.name = 'Christian Bale' AND

m.title = 'Batman Begins' AND

NOT exists((a)-[:ACTED\_IN]->(m))

CREATE (a)-[rel:ACTED\_IN]->(m)

SET rel.roles = ['Bruce Wayne','Batman']

RETURN a, rel, m

|  |  |
| --- | --- |
| NOTE | You can prevent duplication of relationships by merging data using the MERGE clause, rather than the CREATE clause. You will learn about merging data later in this module. |

### Removing properties from a relationship

There are two ways that you can remove a property from a node. One way is to use the REMOVE keyword. The other way is to set the property’s value to null, just as you do for properties of nodes.

Suppose we have added the ACTED\_IN relationship between Christian Bale and the movie, Batman Returns where the roles property is added to the relationship. Here is an example to remove the roles property, yet keep the ACTED\_IN relationship:

MATCH (a:Person)-[rel:ACTED\_IN]->(m:Movie)

WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'

REMOVE rel.roles

RETURN a, rel, m

Here is the result returned. An alternative to REMOVE rel.roles would be SET rel.roles = null

### ****Exercise 9: Creating Relationships****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 9.

1)

MATCH (m:Movie)

WHERE m.title = 'Forrest Gump'

MATCH (p:Person)

WHERE p.name = 'Tom Hanks' OR p.name = 'Robin Wright' OR p.name = 'Gary Sinise'

CREATE (p)-[:ACTED\_IN]->(m)

2)

MATCH (m:Movie)

WHERE m.title = 'Forrest Gump'

MATCH (p:Person)

WHERE p.name = 'Robert Zemeckis'

CREATE (p)-[:DIRECTED]->(m)

3)

MATCH (p1:Person)

WHERE p1.name = 'Tom Hanks'

MATCH (p2:Person)

WHERE p2.name = 'Gary Sinise'

CREATE (p1)-[:HELPED]->(p2)

4)

MATCH (p:Person)-[rel]-(m:Movie)

WHERE m.title = 'Forrest Gump'

RETURN p, rel, m

5)

MATCH (p:Person)-[rel:ACTED\_IN]->(m:Movie)

WHERE m.title = 'Forrest Gump'

SET rel.roles =

CASE p.name

WHEN 'Tom Hanks' THEN ['Forrest Gump']

WHEN 'Robin Wright' THEN ['Jenny Curran']

WHEN 'Gary Sinise' THEN ['Lieutenant Dan Taylor']

END

6)

MATCH (p1:Person)-[rel:HELPED]->(p2:Person)

WHERE p1.name = 'Tom Hanks' AND p2.name = 'Gary Sinise'

SET rel.research = 'war history'

7)

call db.propertyKeys

8)

CALL db.schema.visualization()

9)

MATCH (p:Person)-[rel:ACTED\_IN]->(m:Movie)

WHERE m.title = 'Forrest Gump'

RETURN p.name, rel.roles

10)

MATCH (p1:Person)-[rel:HELPED]-(p2:Person)

RETURN p1.name, rel, p2.name

11)

MATCH (p:Person)-[rel:ACTED\_IN]->(m:Movie)

WHERE m.title = 'Forrest Gump' AND p.name = 'Gary Sinise'

SET rel.roles =['Lt. Dan Taylor']

12)

MATCH (a:Person)-[rel:HELPED]->(b:Person)

WHERE a.name = 'Tom Hanks' AND b.name = 'Gary Sinise'

REMOVE rel.research

RETURN a, rel, b

13)

MATCH (p:Person)-[rel:ACTED\_IN]->(m:Movie)

WHERE m.title = 'Forrest Gump'

return p, rel, m

### Deleting nodes and relationships

If a node has no relationships to any other nodes, you can simply delete it from the graph using the DELETE clause. Relationships are also deleted using the DELETE clause.

|  |  |
| --- | --- |
| NOTE | If you attempt to delete a node in the graph that has relationships in or out of the node, the graph engine will return an error because deleting such a node will leave orphaned relationships in the graph. |

### Deleting relationships

You can delete a relationship between nodes by first finding it in the graph and then deleting it.

In this example, we want to delete the *ACTED\_IN* relationship between *Christian Bale* and the movie *Batman Begins*. We find the relationship, and then delete it:

MATCH (a:Person)-[rel:ACTED\_IN]->(m:Movie)

WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'

DELETE rel

RETURN a, m

Notice that there no longer exists the relationship between *Christian Bale* and the movie *Batman Begins*.

We can now query the nodes related to *Batman Begins* to see that this movie now only has two actors and one producer connected to it:

Even though we have deleted the relationship between actor, Christian Bale and the movie Batman Begins, we note that this actor is connected to another movie in the graph, so we should not delete this Christian Bale node.

In this example, we find the node for the producer, Benjamin Melniker, as well as his relationship to movie nodes. First, we delete the relationship(s), then we delete the node:

MATCH (p:Person)-[rel:PRODUCED]->(:Movie)

WHERE p.name = 'Benjamin Melniker'

DELETE rel, p

### Deleting nodes and relationships

The most efficient way to delete a node and its corresponding relationships is to specify DETACH DELETE. When you specify DETACH DELETE for a node, the relationships to and from the node are deleted, then the node is deleted.

If were were to attempt to delete the Liam Neeson node without first deleting its relationships:

MATCH (p:Person)

WHERE p.name = 'Liam Neeson'

DELETE p

We would see this error:

#### Neo.ClientError.Schema.ConstraintValidationFailed

Cannot delete node<171>, because it still has relationships. To delete this node, you must first delete its relationships.

Here we delete the Liam Neeson node and its relationships to any other nodes:

MATCH (p:Person)

WHERE p.name = 'Liam Neeson'

DETACH DELETE p

And here is what the Batman Begins node and its relationships now look like. There is only one actor, Michael Caine connected to the movie.

### ****Exercise 10: Deleting Nodes and Relationships****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 10.

1)

MATCH (:Person)-[rel:HELPED]-(:Person)

DELETE rel

2)

MATCH (:Person)-[rel:HELPED]-(:Person)

RETURN rel

3)

MATCH (p:Person)-[rel]-(m:Movie)

WHERE m.title = 'Forrest Gump'

RETURN p, rel, m

4)

MATCH (m:Movie)

WHERE m.title = 'Forrest Gump'

DELETE m

5)

MATCH (m:Movie)

WHERE m.title = 'Forrest Gump'

DETACH DELETE m

6)

MATCH (p:Person)-[rel]-(m:Movie)

WHERE m.title = 'Forrest Gump'

RETURN p, rel, m

### Merging data in the graph

Thus far, you have learned how to create nodes, labels, properties, and relationships in the graph. You can use MERGE to either create new nodes and relationships or to make structural changes to existing nodes and relationships.

For example, how the graph engine behaves when a duplicate element is created depends on the type of element:

|  |  |
| --- | --- |
| **If you use CREATE:** | **The result is:** |
| Node | If a node with the same property values exists, a duplicate node is created. |
| Label | If the label already exists for the node, the node is not updated. |
| Property | If the node or relationship property already exists, it is updated with the new value.   **Note:** If you specify a set of properties to be created using = rather than +=, existing properties are removed if they are not included in the set. |
| Relationship | If the relationship exists, a duplicate relationship is created. |

|  |  |
| --- | --- |
| WARNING | You should never create duplicate nodes or relationships in a graph. |

The MERGE clause is used to find elements in the graph. If the element is not found, it is created.

|  |
| --- |
| You use the MERGE clause to:   * Create a unique node based on label and key information for a property and if it exists, optionally update it. * Create a unique relationship. * Create a node and relationship to it uniquely in the context of another node. |

### Using MERGE to create nodes

Here is the simplified syntax for the MERGE clause for creating a node:

MERGE (variable:Label{nodeProperties})

RETURN variable

If there is an existing node with Label and nodeProperties found in the graph, no node is created. If, however the node is not found in the graph, then the node is created.

When you specify nodeProperties for MERGE, you should only use properties that satisfy some sort of uniqueness constraint. You will learn about uniqueness constraints in the next module.

Here is what we currently have in the graph for the Person, Michael Caine. This node has values for name and born. Notice also that the label for the node is Person.

Here we use MERGE to find a node with the Actor label with the key property name of Michael Caine, and we set the born property to 1933. Our data model has never used the label, Actor so this is a new entity type in our graph.

MERGE (a:Actor {name: 'Michael Caine'})

SET a.born = 1933

RETURN a

Here is the result of running this Cypher example. We do not find a node with the label Actor so the graph engine creates one.

|  |  |
| --- | --- |
| NOTE | When you specify the node to merge, you should only use properties that have a unique index. You will learn about uniqueness later in this training. |

If we were to repeat this MERGE clause, no additional *Actor* nodes would be created in the graph.

At this point, however, we have two *Michael Caine* nodes in the graph, one of type *Person*, and one of type *Actor*:

### Using MERGE to create relationships

Here is the syntax for the MERGE clause for creating relationships:

MERGE (variable:Label {nodeProperties})-[:REL\_TYPE]->(otherNode)

RETURN variable

If there is an existing node with Label and nodeProperties with the :REL\_TYPE to otherNode found in the graph, no relationship is created. If the relationship does not exist, it is created.

Although, you can leave out the direction of the relationship being created with the MERGE, in which case a left-to-right arrow will be assumed, a best practice is to always specify the direction of the relationship. However, if you have bidirectional relationships and you want to avoid creating duplicate relationships, you must leave off the arrow.

### Specifying creation behavior when merging

You can use the MERGE clause, along with ON CREATE to assign specific values to a node being created as a result of an attempt to merge.

Here is an example of creating a new node while specifying property values for the new node:

MERGE (a:Person {name: 'Sir Michael Caine'})

ON CREATE SET a.birthPlace = 'London',

a.born = 1934

RETURN a

We know that there are no existing Sir Michael Caine Person nodes. When the MERGE executes, it will not find any matching nodes so it will create one and will execute the ON CREATE clause where we set the birthplace and born property values.

Here are the resulting nodes that have anything to do with Michael Caine. The most recently created node has the name value of Sir Michael Caine.

You can also specify an ON MATCH clause during merge processing. If the exact node is found, you can update its properties or labels. Here is an example:

MERGE (a:Person {name: 'Sir Michael Caine'})

ON CREATE SET a.born = 1934,

a.birthPlace = 'UK'

ON MATCH SET a.birthPlace = 'UK'

RETURN a

And here we see that the found node (with the <id> of 1920) was updated with the new value for birthPlace.

### Using MERGE to create relationships

Using MERGE to create relationships is expensive and you should only do it when you need to ensure that a relationship is unique and you are not sure if it already exists.

In this example, we use the MATCH clause to find all Person nodes that represent Michael Caine and we find the movie, Batman Begins that we want to connect to all of these nodes. We already have a connection between one of the Person nodes and the Movie node. We do not want this relationship to be duplicated. This is where we can use MERGE as follows:

MATCH (p:Person), (m:Movie)

WHERE m.title = 'Batman Begins' AND p.name ENDS WITH 'Caine'

MERGE (p)-[:ACTED\_IN]->(m)

RETURN p, m

Here is the result of executing this Cypher statement. It went through all the nodes and added the relationship to the nodes that didn’t already have the relationship.

You must be aware of the behavior of the MERGE clause and how it will automatically create nodes and relationships. MERGE tries to find a full pattern and if it doesn’t find it, it creates that full pattern. That’s why in most cases you should first MERGE your nodes and then your relationship afterwards.

Only if you intentionally want to create a node within the context of another (like a month within a year) then a MERGE pattern with one bound and one unbound node makes sense.

For example:

MERGE (fromDate:Date {year: 2018})<-[:IN\_YEAR]-(toDate:Date {month: 'January'})

### ****Exercise 11: Merging Data in the graph****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 11.

1)

MERGE (m:Movie {title: 'Forrest Gump'})

ON CREATE SET m.released = 1994

RETURN m

2)

MERGE (m:Movie {title: 'Forrest Gump'})

ON CREATE SET m.released = 1994

ON MATCH SET m.tagline = "Life is like a box of chocolates...you never know what you're gonna get."

RETURN m

3)

MERGE (p:Production {title: 'Forrest Gump'})

ON CREATE SET p.year = 1994

RETURN p

4)

MATCH (m)

WHERE m.title = 'Forrest Gump'

RETURN labels(m)

5)

MERGE (p:Production {title: 'Forrest Gump'})

ON MATCH SET p.company = 'Paramount Pictures'

RETURN p

6)

MERGE (m:Movie {title: 'Forrest Gump'})

ON MATCH SET m:OlderMovie

RETURN labels(m)

7)

MERGE (p:Person {name: 'Robert Zemeckis'})-[:DIRECTED]->(m {title: 'Forrest Gump'})

This statement above first finds all Person nodes that have only the name property value of Robert Zemeckis. It then finds all nodes with only the title property set to Forrest Gump. There are no Person or other nodes that have only these properties so the graph engine creates them. Then the graph engine creates the relationship between these two nodes. That is, this MERGE operation creates two nodes and a single relationship. If we had provided all of the property values for the nodes, we would not have created the extra nodes.

In fact, you should **never** create nodes and relationships together like this! This example is here to show you how powerful Cypher can be. A best practice is to create nodes first, then relationships.

8)

MERGE (p:Person {name: 'Robert Zemeckis'})-[:DIRECTED]->(m {title: 'Forrest Gump'})

9)

query the nodes before you delete them to ensure you have the correct MATCH clauses. Execute this query:

MATCH (p:Person {name: 'Robert Zemeckis'})-[rel]-(x)

WHERE NOT EXISTS (p.born)

RETURN p, rel, x

10)

MATCH (p:Person {name: 'Robert Zemeckis'})--()

WHERE NOT EXISTS (p.born)

DETACH DELETE p

11)

MATCH (m)

WHERE m.title = 'Forrest Gump' AND labels(m) = []

RETURN m, labels(m)

12)

MATCH (m)

WHERE m.title = 'Forrest Gump' AND labels(m) = []

DETACH DELETE m

13)

MATCH (p:Person), (m:Movie)

WHERE p.name = 'Robert Zemeckis' AND m.title = 'Forrest Gump'

MERGE (p)-[:DIRECTED]->(m)

14)

MATCH (p:Person), (m:Movie)

WHERE p.name IN ['Tom Hanks','Gary Sinise', 'Robin Wright']

AND m.title = 'Forrest Gump'

MERGE (p)-[:ACTED\_IN]->(m)

15)

MATCH (p:Person)-[rel:ACTED\_IN]->(m:Movie)

WHERE m.title = 'Forrest Gump'

SET rel.roles =

CASE p.name

WHEN 'Tom Hanks' THEN ['Forrest Gump']

WHEN 'Robin Wright' THEN ['Jenny Curran']

WHEN 'Gary Sinise' THEN ['Lt. Dan Taylor']

END

MODULE 7

# **Getting More Out of Neo4j**

### About this module

You have learned how to set up your development environment for accessing a Neo4j graph and how to write basic Cypher statements for querying the graph modifying the graph.

At the end of this module, you should be able to:

* Use parameters in your Cypher statements.
* Analyze Cypher execution.
* Monitor queries.
* Manage constraints and node keys for the graph.
* Import data into a graph from CSV files.
* Manage indexes for the graph.
* Access Neo4j resources.

Cypher parameters

In a deployed application, you should not **hard code** values in your Cypher statements. You use a variety values when you are testing your Cypher statements. But you don’t want to change the Cypher statement every time you test. In addition, you typically include Cypher statements in an application where parameters are passed in to the Cypher statement before it executes. For these scenarios, you should parameterize values in your Cypher statements.

Using Cypher parameters

In your Cypher statements, a parameter name begins with the $ symbol.

Here is an example where we have parameterized the query:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = $actorName

RETURN m.released, m.title ORDER BY m.released DESC

At runtime, if the parameter $actorName has a value, it will be used in the Cypher statement when it runs in the graph engine.

In Neo4j Browser, you can set values for Cypher parameters that will be in effect during your session.

You can set the value of a single parameter in the query editor pane as shown in this example where the value *Tom Hanks* is set for the parameter actorName:

:param actorName => 'Tom Hanks'

|  |  |
| --- | --- |
| NOTE | You can even specify a Cypher expression to the right of => to set the value of the parameter. |

Notice here that :param is a client-side browser command. It takes a name and expression and stores the value of that expression for the name in the session.

After the *actorName* parameter is set, you can run the query that uses the parameter:

Subsequently, you need only change the value of the parameter and not the Cypher statement to test with different values.

After we have changed the *actorName* parameter to ‘Tom Cruise’, we get a different result with the same Cypher query:

You can also use the JSON-style syntax to set all of the parameters in your Neo4j Browser session. The values you can specify in this object are numbers, strings, and booleans. In this example we set two parameters for our session:

:params {actorName: 'Tom Cruise', movieName: 'Top Gun'}

If you want to remove an existing parameter from your session, you do so by using the JSON-style syntax and excluding the parameter for your session.

If you want to view the current parameters and their values, simply type :params:

If you want to clear all parameters, you can simply type:

:params {}

### ****Exercise 12: Using Cypher parameters****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 12.

1)

MATCH (r:Person)-[rel:REVIEWED]->(m:Movie)<-[:ACTED\_IN]-(a:Person)

RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)

2)

:param year => 2000

3)

MATCH (r:Person)-[rel:REVIEWED]->(m:Movie)<-[:ACTED\_IN]-(a:Person)

WHERE m.released = $year

RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)

4)

:param year => 2006

5)

:params {year: 2006, ratingValue: 65}

6)

MATCH (r:Person)-[rel:REVIEWED]->(m:Movie)<-[:ACTED\_IN]-(a:Person)

WHERE m.released = $year AND

rel.rating > $ratingValue

RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)

7)

:params {year: 2006, ratingValue: 60}

### Analyzing Cypher execution

The *Movie* graph that you have been using during training is a very small graph. As you start working with large datasets, it will be important to not only add appropriate indexes to your graph, but also write Cypher statements that execute as efficiently as possible.

There are two Cypher keywords you can prefix a Cypher statement with to analyze a query:

* EXPLAIN provides estimates of the graph engine processing that will occur, but does not execute the Cypher statement.
* PROFILE provides real profiling information for what has occurred in the graph engine during the query and executes the Cypher statement.

The EXPLAIN option provides the Cypher query plan. You can compare different Cypher statements to understand the stages of processing that will occur when the Cypher executes.

Here is an example where we have set the *actorName* and *year* parameters for our session and we execute this Cypher statement:

EXPLAIN MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = $actorName AND

m.released < $year

RETURN p.name, m.title, m.released

You can expand each phase of the Cypher execution to examine what code is expected to run. Each phase of the query presents you with an estimate of the number of rows expected to be returned. With EXPLAIN, the query does not run, the graph engine simply produces the query plan.

For a better metric for analyzing how the Cypher statement will run you use the PROFILE keyword which runs the Cypher statement and gives you run-time performance metrics.

Here we see that for each phase of the graph engine processing, we can view the cache hits and most importantly the number of times the graph engine accessed the database (db hits). This is an important metric that will affect the performance of the Cypher statement at run-time.

For example, if we were to change the Cypher statement so that the node labels are not specified, we see these metrics when we profile:

Here we see more db hits which makes sense because all nodes need to be scanned for perform this query.

Monitoring queries

If you are testing an application and have run several queries against the graph, there may be times when your Neo4j Browser session hangs with what seems to be a very long-running query. There are two reasons why a Cypher query may take a long time:

* The query returns a lot of data. The query completes execution in the graph engine, but it takes a long time to create the result stream.
  + Example: MATCH (a)--(b)--(c)--(d)--(e)--(f) RETURN a
* The query takes a long time to execute in the graph engine.
  + Example: MATCH (a), (b), (c), (d), (e) RETURN count(id(a))

If the query executes and then **returns a lot of data**, there is no way to monitor it or kill the query. All that you can do is close your Neo4j Browser session and start a new one. If the server has many of these **rogue** queries running, it will slow down considerably so you should aim to limit these types of queries. If you are running Neo4j Desktop, you can simply restart the database to clear things up, but if you are using a Neo4j Sandbox, you cannot do so. The database server is always running and you cannot restart it. Your only option is to shut down the Neo4j Sandbox and create a new Neo4j Sandbox, but then you lose any data you have worked with.

If, however, the query is a **long-running query**, you can monitor it by using the :queries command. Here is a screenshot where we are monitoring a long-running query in another Neo4j Browser session:

The :queries command calls dbms.listQueries under the hood, which is why we see two queries here. We have turned on **AUTO-REFRESH** so we can monitor the number of ms used by the graph engine thus far. You can kill the running query by double-clicking the icon in the Kill column. Alternatively, you can execute the statement CALL dbms.killQuery('query-id').

|  |  |
| --- | --- |
| NOTE | The :queries command is only available in the Enterprise Edition of Neo4j. |

### ****Exercise 13: Analyzing and monitoring queries****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 13.

1)

:params

--------------

MATCH (r:Person)-[rel:REVIEWED]->(m:Movie)<-[:ACTED\_IN]-(a:Person)

WHERE m.released = $year AND

rel.rating > $ratingValue

RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)

--------------

EXPLAIN MATCH (r:Person)-[rel:REVIEWED]->(m:Movie)<-[:ACTED\_IN]-(a:Person)

WHERE m.released = $year AND

rel.rating > $ratingValue

RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)

2)

PROFILE MATCH (r:Person)-[rel:REVIEWED]->(m:Movie)<-[:ACTED\_IN]-(a:Person)

WHERE m.released = $year AND

rel.rating > $ratingValue

RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)

3)

PROFILE MATCH (r)-[rel]->(m)<-[:ACTED\_IN]-(a)

WHERE m.released = $year AND

rel.rating > $ratingValue

RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)

4)

open a Web browser window and enter [http://localhost:7474](http://localhost:7474/) which opens a second Neo4j Browser window that has access to the same graph.

5)

PROFILE MATCH (a)--(b)--(c)--(d)--(e)--(f)--(g)

RETURN a

------------------------

IN THE SECOND BROWSER => :queries

6)

| **Database URI** | **User** | **Query** | **Params** | **Meta** | **Elapsed time** | **Kill** |
| --- | --- | --- | --- | --- | --- | --- |
| neo4j://localhost:7687 | neo4j | CALL dbms.listQueries | {} | {} | 1 ms |  |
| neo4j://localhost:7687 | neo4j | PROFILE MATCH (a), (b), (c), (d), (e) , (f), (g) RETURN count(id(a)) | { "year": "2006.0", "ratingValue": "60.0" } | {} | 111633 ms |  |

7)

In the second Neo4j Browser window, kill the long-running query.

Managing constraints and node keys

You have seen that you can accidentally create duplicate nodes in the graph if you’re not protected. In most graphs, you will want to prevent duplication of data. Unfortunately, you cannot prevent duplication by checking the existence of the exact node (with properties) as this type of test is not cluster or multi-thread safe as no locks are used. This is one reason why MERGE is preferred over CREATE, because MERGE does use locks.

In addition, you have learned that a node or relationship need not have a particular property. What if you want to ensure that all nodes or relationships of a specific type (label) must set values for certain properties?

A third scenario with graph data is where you want to ensure that a set of property values for nodes of the same type, have a unique value. This is the same thing as a primary key in a relational database.

All of these scenarios are common in many graphs. In Neo4j, you can use Cypher to:

* Add a *uniqueness constraint* that ensures that a value for a property is unique for all nodes of that type.
* Add an *existence constraint* that ensures that when a node or relationship is created or modified, it must have certain properties set.
* Add a *node key* that ensures that a set of values for properties of a node of a given type is unique.

Constraints and node keys that enforce uniqueness are related to indexes which you will learn about later in this module.

|  |  |
| --- | --- |
| NOTE | Existence constraints and node keys are only available in Enterprise Edition of Neo4j. |

### Ensuring that a property value for a node is unique

You add a uniqueness constraint to the graph by creating a constraint that asserts that a particular node property is unique in the graph for a particular type of node.

Here is an example for ensuring that the title for a node of type Movie is unique:

CREATE CONSTRAINT ON (m:Movie) ASSERT m.title IS UNIQUE

This Cypher statement will fail if the graph already has multiple Movie nodes with the same value for the title property. Note that you can create a uniqueness constraint, even if some Movie nodes do not have a title property.

And if we attempt to create a *Movie* with the *title*, *The Matrix*, the Cypher statement will fail because the graph already has a movie with that title:

CREATE (:Movie {title: 'The Matrix'})

Here is the result of running this Cypher statement on the *Movie* graph:

#### Neo.ClientError.Schema.ConstraintValidationFailed

Node(5) already exists with label `Movie` and property `title` = 'The Matrix'

In addition, if you attempt to modify the value of a property where the uniqueness assertion fails, the property will not be updated.

### Ensuring that properties exist

Having uniqueness for a property value is only useful in the graph if the property exists. In most cases, you will want your graph to also enforce the existence of properties, not only for those node properties that require uniqueness, but for other nodes and relationships where you require a property to be set. Uniqueness constraints can only be created for nodes, but existence constraints can be created for node or relationship properties.

You add an existence constraint to the graph by creating a constraint that asserts that a particular type of node or relationship property must exist in the graph when a node or relationship of that type is created or updated.

Recall that in the Movie graph, the movie, Something’s Gotta Give has no tagline property:

Here is an example for adding the existence constraint to the *tagline* property of all *Movie* nodes in the graph:

CREATE CONSTRAINT ON (m:Movie) ASSERT exists(m.tagline)

The constraint cannot be added to the graph because a node has been detected that violates the constraint.

We know that in the *Movie* graph, all *:REVIEWED* relationships currently have a property, *rating*. We can create an existence constraint on that property as follows:

CREATE CONSTRAINT ON ()-[rel:REVIEWED]-() ASSERT exists(rel.rating)

Notice that when you create the constraint on a relationship, you need not specify the direction of the relationship.

So after creating this constraint, if we attempt to create a *:REVIEWED* relationship without setting the *rating* property:

MATCH (p:Person), (m:Movie)

WHERE p.name = 'Jessica Thompson' AND

m.title = 'The Matrix'

MERGE (p)-[:REVIEWED {summary: 'Great movie!'}]->(m)

We see this error:

#### Neo.ClientError.Schema.ConstraintValidationFailed

Relationship(5) with type `REVIEWED` must have the property `rating`

You will also see this error if you attempt to remove a property from a node or relationship where the existence constraint has been created in the graph.

### Retrieving constraints defined for the graph

You can run the browser command :schema to view existing indexes and constraints defined for the graph.

Just as you have used other db related methods to query the schema of the graph, you can query for the set of constraints defined in the graph as follows:

CALL db.constraints()

And here is what is returned from the graph:

|  |  |
| --- | --- |
| NOTE | Using the method notation for the CALL statement enables you to use the call for returning results that may be used later in the Cypher statement. |

### Dropping constraints

You use similar syntax to drop an existence or uniqueness constraint, except that you use the DROP keyword rather than CREATE

Here we drop the existence constraint for the rating property for all REVIEWED relationships in the graph:

DROP CONSTRAINT ON ()-[rel:REVIEWED]-() ASSERT exists(rel.rating)

### Creating node keys

A node key is used to define the uniqueness constraint for multiple properties of a node of a certain type. A node key is also used as a composite index in the graph.

Suppose that in our Movie graph, we will not allow a Person node to be created where both the name and born properties are the same. We can create a constraint that will be a node key to ensure that this uniqueness for the set of properties is asserted.

Here is an example to create this node key:

CREATE CONSTRAINT ON (p:Person) ASSERT (p.name, p.born) IS NODE KEY

Here is the result of running this Cypher statement on our Movie graph:

#### Neo.DatabaseError.Schema.ConstraintCreationFailed

Unable to create CONSTRAINT ON ( person:Person ) ASSERT exists(person.name, person.born):

Node(134) with label `Person` must have the properties (name, born)

This attempt to create the constraint failed because there are *Person* nodes in the graph that do not have the *born* property defined.

If we set these properties for all nodes in the graph that do not have *born* properties with:

MATCH (p:Person)

WHERE NOT exists(p.born)

SET p.born = 0

Then the creation of the node key succeeds

Any subsequent attempt to create or modify an existing *Person* node with *name* or *born* values that violate the uniqueness constraint as a node key will fail.

For example, executing this Cypher statement will fail:

CREATE (:Person {name: 'Jessica Thompson', born: 0})

### ****Exercise 14: Managing constraints and node keys****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 14.

1)

CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE

2)

CREATE (:Person {name: 'Tom Hanks'})

3)

CREATE CONSTRAINT ON (p:Person) ASSERT exists(p.born)

4)

MATCH (p:Person)

WHERE NOT exists(p.born)

SET p.born = 0

5)

CREATE CONSTRAINT ON (p:Person) ASSERT exists(p.born)

6)

CREATE (:Person {name: 'Sean Penn'})

7)

CREATE CONSTRAINT ON ()-[r:ACTED\_IN]-() ASSERT exists(r.roles)

8)

MATCH (p:Person), (m:Movie)

WHERE p.name = 'Emil Eifrem' AND

m.title = 'Forrest Gump'

MERGE (p)-[:ACTED\_IN]->(m)

9)

CREATE CONSTRAINT ON (m:Movie) ASSERT (m.title, m.released) IS NODE KEY

10)

CREATE (:Movie {title: 'Back to the Future', released: 1985, tagline: 'Our future.'})

11)

CREATE (:Movie {title: 'Back to the Future', released: 2018, tagline: 'The future is ours.'})

12)

CREATE (:Movie {title: 'Back to the Future', released: 2018, tagline: 'The future is ours.'})

13)

CALL db.constraints()

14)

DROP CONSTRAINT ON ()-[ acted\_in:ACTED\_IN ]-() ASSERT exists(acted\_in.roles)

### Managing indexes

The uniqueness and node key constraints that you add to a graph are essentially single-property and composite indexes respectively. Indexes are used to improve initial node lookup performance, but they require additional storage in the graph to maintain and also add to the cost of creating or modifying property values that are indexed. Indexes store redundant data that points to nodes with the specific property value or values. Unlike SQL, there is no such thing as a primary key in Neo4j. You can have multiple properties on nodes that must be unique.

Here is a brief summary of when single-property indexes are used:

* Equality checks =
* Range comparisons >,>=,<, <=
* List membership IN
* String comparisons STARTS WITH, ENDS WITH, CONTAINS
* Existence checks exists()
* Spatial distance searches distance()
* Spatial bounding searches point()

Composite indexes are used only for equality checks and list membership.

In this module, we introduce the basics of Neo4j indexes, but you should consult the Neo4j *Operations Manual* for more details about creating and maintaining indexes.

|  |  |
| --- | --- |
| NOTE | Because index maintenance incurs additional overhead when nodes are created, We recommend that for large graphs, indexes are created after the data has been loaded into the graph. You can view the progress of the creation of an index when you use the :schema command. |

### Indexes for range searches

When you add an index for a property of a node, it can greatly reduce the number of nodes the graph engine needs to visit in order to satisfy a query.

In this query we are testing the value of the released property of a Movie node using ranges:

MATCH (m:Movie)

WHERE 1990 < m.released < 2000

SET m.videoFormat = 'DVD'

The graph engine, using an index, will find the pointers to all nodes that satisfy the query without having to visit all of the nodes:

### Creating indexes

You create an index to improve graph engine performance. A unique constraint on a property is an index so you need not create an index for any properties you have created uniqueness constraints for. An index on its own does not guarantee uniqueness.

Here is an example of how we would create a single-property index on the released property of all nodes of type Movie:

CREATE INDEX ON :Movie(released)

If a set of properties for a node must be unique for every node, then you should create a constraint as a node key, rather than an index.

If, however, there can be duplication for a set of property values, but you want faster access to them, then you can create a composite index. A composite index is based upon multiple properties for a node.

Suppose we added the property, *videoFormat* to every *Movie* node and set its value, based upon the released date of the movie as follows:

MATCH (m:Movie)

WHERE m.released >= 2000

SET m.videoFormat = ['DVD','BlueRay'];

MATCH (m:Movie)

WHERE m.released < 2000

SET m.videoFormat = ['VHS','DVD']

|  |  |
| --- | --- |
| NOTE | Notice that in the above Cypher statements we use the semi-colon ; to separate Cypher statements. In general, you need not end a Cypher statement with a semi-colon, but if you want to execute multiple Cypher statements, you must separate them. You have already used the semi-colon to separate Cypher statements when you loaded the *Movie* database in the training exercises. |

Now that the graph has *Movie* nodes with both the properties, *released* and *videoFormat*, we can create a composite index on these properties as follows:

CREATE INDEX ON :Movie(released, videoFormat)

### Retrieving indexes

Just as you can retrieve the constraints defined for the graph using :schema or CALL db.constraints(), you can retrieve the indexes:

CALL db.indexes()

Notice that the unique constraints and node keys are also shown as indexes in the graph.

### Dropping indexes

You can drop an existing index that you created with CREATE INDEX.

Here is an example of dropping the composite index that we just created:

DROP INDEX ON :Movie(released, videoFormat)

### ****Exercise 15: Managing indexes****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 15.

1)

CREATE INDEX ON :Person(born)

2)

CALL db.indexes()

3)

DROP INDEX ON :Person(born)

### Going from relational to graph with Neo4j

In this video, you will learn how developers use Neo4j for implementing all or part of their relational models.

### Importing data

In many applications, it is the case that the data that you want to populate your graph with comes from data that was written to .csv files or files of other types. There are many nuances and best practices for loading data into a graph from files. In this module, you will be introduced to some simple steps for loading CSV data into your graph with Cypher. If you are interested in direct loading of data from a relational DBMS into a graph, you should read about the Neo4j Extract Transform Load (ETL) tool at <http://neo4j.com/developer/neo4j-etl/>, as well as many of the useful pre-written procedures that are available for your use in the APOC library.

In Cypher, you can:

* Load data from a URL (http(s) or file).
* Process data as a stream of records.
* Create or update the graph with the data being loaded.
* Use transactions during the load.
* Transform and convert values from the load stream.
* Load up to 10M nodes and relationships.

CSV import is commonly used to import data into a graph. If you want to import data from CSV, you will need to first develop a model that describes how data from your CSV maps to data in your graph.

### Importing normalized data using LOAD CSV

Cypher provides an elegant built-in way to import tabular CSV data into graph structures.

The LOAD CSV clause parses a local file in the **import** directory of your Neo4j installation or a remote file into a stream of rows which represent maps (with headers) or lists. Then you can use whichever Cypher operations you want to either create nodes or relationships or to merge with the existing graph.

Here is the simplified syntax for using LOAD CSV:

LOAD CSV WITH HEADERS FROM url-value

AS row // row is a variable that is used to extract data

The first line of the file must contain a comma-separated list of column names. The url-value can be a resource or a file on your system. Each line contains data that is interpreted as values for each column name. When each line is read from the file, you can perform the necessary processing to create or merge data into the graph.

As CSV files usually represent either node or relationship lists, you will run multiple passes to create nodes and relationships separately.

The **movies\_to\_load.csv** file (sample below) contains the data that will add Movie nodes:

id,title,country,year,summary

1,Wall Street,USA,1987, Every dream has a price.

2,The American President,USA,1995, Why can't the most powerful man in the world have the one thing he wants most?

3,The Shawshank Redemption,USA,1994, Fear can hold you prisoner. Hope can set you free.

Before you load data from CSV files into your graph, you should first confirm that the data retrieved looks OK. Rather than creating nodes or relationships, you can simply return information about the data to be loaded.

For example you can execute this Cypher statement to get a count of the data to be loaded from the **movies\_to\_load.csv** file so you have an idea of how much data will be loaded:

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies\_to\_load.csv'

AS line

RETURN count(\*)

You might even want to visually inspect the data before you load it to see if it is what you were expecting:

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/intro-neo4j/movies\_to\_load.csv'

AS line

RETURN \* LIMIT 1

Notice here that the *summary* column’s data has an extra space before the data in the file. In order to ensure that all *tagline* values in our graph do not have an extra space, we will trim the value before assigning it to the tagline property. Once we are sure you want to load the data into your graph, we do so by assigning values from each row read in to a new node.

You may want to format the data before it is loaded to confirm it matches what you want in your graph:

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies\_to\_load.csv'

AS line

RETURN line.id, line.title, toInteger(line.year), trim(line.summary)

The following query creates the *Movie* nodes using some of the data from **movies\_to\_load.csv** as properties:

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/intro-neo4j/movies\_to\_load.csv'

AS line

CREATE (movie:Movie { movieId: line.id, title: line.title, released: toInteger(line.year) , tagline: trim(line.summary)})

We assign a value to *movieId* from the *id* data in the CSV file. In addition, we assign the data from *summary* to the *tagline* property, with a trim. We also convert the data read from *year* to an integer using the built-in function toInteger() before assigning it to the *released* property.

The **persons\_to\_load.csv** file (sample below) holds the data that will populate the *Person* nodes.

Id,name,birthyear

1,Charlie Sheen, 1965

2,Oliver Stone, 1946

3,Michael Douglas, 1944

4,Martin Sheen, 1940

5,Morgan Freeman, 1937

In case you already have people in your database, you will want to avoid creating duplicates. That’s why instead of just creating them, we use MERGE to ensure unique entries after the import. We use the ON CREATE clause to set the values for *name* and *born*.

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/intro-neo4j/persons\_to\_load.csv'

AS line

MERGE (actor:Person { personId: line.Id })

ON CREATE SET actor.name = line.name,

actor.born = toInteger(trim(line.birthyear))

There are a couple of things to note here. The name of the column is case-sensitive. In addition, notice that the data for the birthyear column as an extra space before the data. To allow this data to be converted to an integer, we must first trim the whitespace using the trim() built-in function.

The **roles\_to\_load.csv** file (sample below) holds the data that will populate the relationships between the nodes.

personId,movieId,role

1,1,Bud Fox

4,1,Carl Fox

3,1,Gordon Gekko

4,2,A.J. MacInerney

3,2,President Andrew Shepherd

5,3,Ellis Boyd 'Red' Redding

The query below matches the entries of *line.personId* and *line.movieId* to their respective *Movie* and *Person* nodes, and creates an *ACTED\_IN* relationship between the person and the movie. This model includes a relationship property of *role*, which is passed via *line.role*.

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/intro-neo4j/roles\_to\_load.csv'

AS line

MATCH (movie:Movie { movieId: line.movieId })

MATCH (person:Person { personId: line.personId })

CREATE (person)-[:ACTED\_IN { roles: [line.role]}]->(movie)

### Importing denormalized data

If your file contains denormalized data, you can run the same file with multiple passes and simple operations as shown above. Alternatively, you might have to use MERGE to create nodes and relationships uniquely.

For our use case, we can import the data using a CSV structure like this:

**movie\_actor\_roles\_to\_load.csv**:

title;released;summary;actor;birthyear;characters

Back to the Future;1985;17 year old Marty McFly got home early last night. 30 years early.;Michael J. Fox;1961;Marty McFly

Back to the Future;1985;17 year old Marty McFly got home early last night. 30 years early.;Christopher Lloyd;1938;Dr. Emmet Brown

Here are the Cypher statements to load this data:

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/intro-neo4j/movie\_actor\_roles\_to\_load.csv'

AS line FIELDTERMINATOR ';'

MERGE (movie:Movie { title: line.title })

ON CREATE SET movie.released = toInteger(line.released),

movie.tagline = line.summary

MERGE (actor:Person { name: line.actor })

ON CREATE SET actor.born = toInteger(line.birthyear)

MERGE (actor)-[r:ACTED\_IN]->(movie)

ON CREATE SET r.roles = split(line.characters,',')

Notice a couple of things in this Cypher statement. This file uses a semi-colon as a field terminator, rather than the default comma. In addition, the built-in method split() is used to create the list for the roles property.

For large denormalized files, it may still make sense to create nodes and relationships separately in multiple passes. That would depend on the complexity of the operations and the experienced performance.

### Importing a large dataset

If you import a larger amount of data (more than 10,000 rows), it is recommended to prefix your LOAD CSV clause with a PERIODIC COMMIT hint. This allows the database to regularly commit the import transactions to avoid memory churn for large transaction-states.

### ****Exercise 16: Importing data****

In the query edit pane of Neo4j Browser, execute the browser command: :play intro-neo4j-exercises and follow the instructions for Exercise 16.

1)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/actors.csv'

AS line

RETURN line.id, line.name, line.birthYear

2)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/actors.csv'

AS line

RETURN line.id, line.name, toInteger(trim(line.birthYear))

3)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/actors.csv'

AS line

MERGE (actor:Person {name: line.name})

ON CREATE SET actor.born = toInteger(trim(line.birthYear)), actor.actorId = line.id

ON MATCH SET actor.actorId = line.id

4)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies.csv'

AS line

RETURN line.id, line.title, line.year, line.tagLine

5)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies.csv'

AS line

RETURN line.id, line.title, toInteger(line.year), trim(line.tagLine)

6)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies.csv'

AS line

MERGE (m:Movie {title: line.title})

ON CREATE

SET m.released = toInteger(trim(line.year)),

m.movieId = line.id,

m.tagline = line.tagLine

ON MATCH SET m.movieId = line.id

7)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/roles.csv'

AS line FIELDTERMINATOR ';'

RETURN line.personId, line.movieId, line.Role

8)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/roles.csv'

AS line FIELDTERMINATOR ';'

RETURN line.personId, line.movieId, split(line.Role,',')

9)

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/roles.csv'

AS line FIELDTERMINATOR ';'

MATCH (movie:Movie { movieId: line.movieId })

MATCH (person:Person { actorId: line.personId })

MERGE (person)-[:ACTED\_IN { roles: split(line.Role,',')}]->(movie)

### Accessing Neo4j resources

# **Summary: Introduction to Neo4j**

### Quiz Results

All quizes taken successfully.

### Course Completion Certificate

[Download Certificate](https://graphacademy.neo4j.com/training/certificates/e35f45bae1bf9b94a1db232c5d18d176944324fb716cc343f25cabb4dbccf8c1.pdf)

### Course Summary

In this course, you have learned how to:

* Describe what a graph database is.
* Describe how to model relational data in a property graph model.
* Describe the Neo4j Graph Platform and its components.
* Describe the features and benefits of Neo4j.
* Set up your development environment for performing the hands-on exercises of this course which is one of:
  + Neo4j Sandbox
  + Neo4j Desktop
* Use MATCH to retrieve nodes from the graph.
* Use MATCH to retrieve relationships from the graph.
* Use MATCH to retrieve properties from the graph.
* Use the WHERE clause for queries.
* Control query processing.
* Control how results are returned.
* Work with Cypher dates and lists.
* Create, update, and delete nodes and properties of nodes.
* Create, update, and delete relationships and properties of relationships.
* Merge data in the graph.
* Use parameters.
* Define constraints in the graph.
* Profile and monitor query execution.
* Define indexes in the graph.
* Import relational data into the graph.

### Next steps

There are many resources available to you for learning more about Neo4j and Cypher.

Resources for developers:

<https://neo4j.com/developer/resources/>

Neo4j Community Site where you can ask or answer questions about Neo4j and discuss with other users:

[https://community.neo4j.com](https://community.neo4j.com/)

Neo4j documentation:

<https://neo4j.com/docs/>

Neo4j Sandboxes for experimenting with graphs:

<https://neo4j.com/sandbox-v2/>

Videos on YouTube:

<https://www.youtube.com/channel/UCvze3hU6OZBkB1vkhH2lH9Q>

GitHub repository:

<https://github.com/neo4j-contrib>

Neo4j events all over the world:

<https://neo4j.com/events/world/all/>

Neo4j online and classroom training:

<https://neo4j.com/graphacademy/>

Graph Gists for learning more use cases for Neo4j:

<https://neo4j.com/graphgists/>

Become a Neo4j certified developer:

<https://neo4j.com/graphacademy/neo4j-certification/>

|  |  |
| --- | --- |
| NOTE | The certification exam tests you on content from this course. You can take the certification multiple times until you pass! |

Attend a Neo4j meetup:

<https://www.meetup.com/topics/neo4j/>

View questions/answers raised about Neo4j:

<https://stackoverflow.com/tags/neo4j/hot>

### Course feedback

We want your feedback on this course. Please provide your [feedback](https://goo.gl/forms/BEpb5Dsj6CFTvWiI2) so we can improve this course.