**Introduction to Graph Databases**

<https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D>

Table of Contents

* [About this Course](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_about_this_course)
* [Student Survey](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_student_survey)
* [Why use a Graph Database?](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_why_use_a_graph_database)
* [What is Cypher](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_what_is_cypher)
* [The Movie Database](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_the_movie_database)
* [Patterns of Nodes and Relationships](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_patterns_of_nodes_and_relationships)
* [Nodes and their Properties](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_nodes_and_their_properties)
* [Relationships](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_relationships_2)
* [Deleting Nodes](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_deleting_nodes)
* [Lab: All Characters in the Matrix](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_lab_all_characters_in_the_matrix)
* [Order By, Limit, and Skip](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_order_by_limit_and_skip)
* [Predicates and Conditions](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_predicates_and_conditions)
* [Review: Exploring the Movie Database](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_review_exploring_the_movie_database)
* [Matching Paths](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_matching_paths)
* [Returning Paths](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_returning_paths)
* [Indexing and Labels](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_indexing_and_labels)
* [Aggregation](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_aggregation)
* [Lab: Who are the five busiest actors?](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_lab_who_are_the_five_busiest_actors)
* [Lab: Recommendation Engine](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_lab_recommendation_engine)
* [Importing Data](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_importing_data)
* [Community Resources and Next Steps](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_community_resources_and_next_steps)
* [End-of-Course Survey](https://neo4j.com/online_training/graphdatabases/?aliId=IC8%3D#_end_of_course_survey)

**About this Course**

**Motivations**

This training class is an introduction to graph databases. You will learn how to get started with Neo4j using Cypher, the graph query language. Please give us [feedback](mailto:training@neotechnology.com) on how we can improve it.

**Running the Course with Neo4j Browser**

You can run this course also with an [installed and running Neo4j](http://neo4j.com/download) instance.

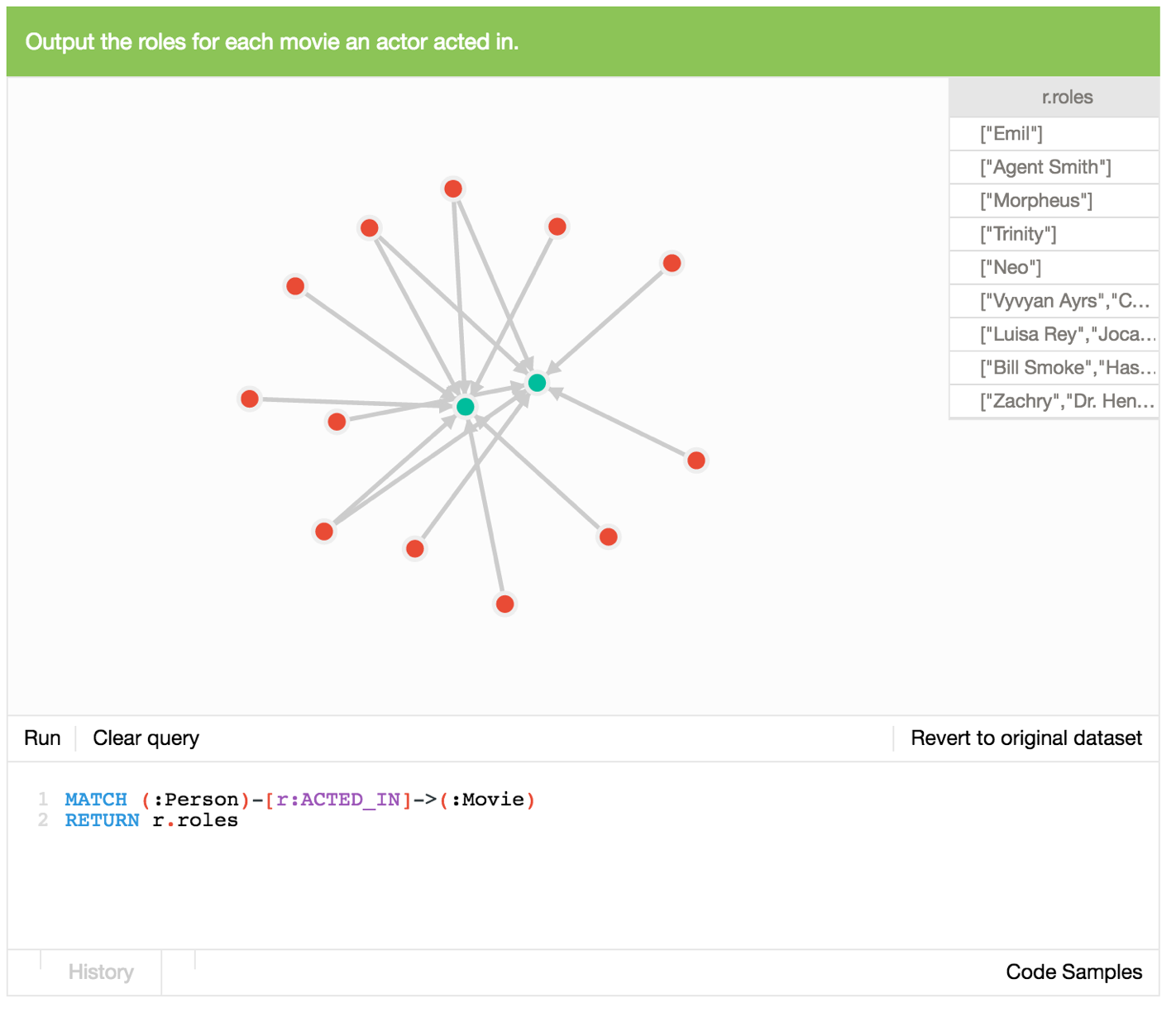
After starting the database, open the Neo4j Browser on <http://localhost:7474/> (if you’re running it locally). Log in. Then run the :play movie graph guide from the command line on top to insert the data and work along with the queries presented in this training.

**The Cypher Query Widget**

If you haven’t installed Neo4j on your machine, the Cypher Query Widget is here for you to interactively explore. You can get comfortable with Cypher and learn how to query graph databases in a safe environment. You can’t break anything, and even if anything happens, just reset the temporary database to its original state.

If you click the "Try it out: …​" buttons, the widget will open in an overlay.

**Example Widget**



The different widgets use different datasets depending on the task, but save your changes across sessions. For most lab exercises we added helpful checks and messages to guide you along.

The query results are shown in a tabular view, and if you return nodes or relationships, those will be highlighted in the graph visualization of the dataset. Complex elements are folded in the result table and can be expanded with a click.

Your history is available across sessions. And we provide some pre-canned queries to get you started quickly.

Please try to work with the tasks on your own and not just copy and paste queries. Thinking about them and writing & executing them incrementally helps you much more in understanding the concepts.

**Student Survey**

Loading...

**Why use a Graph Database?**

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. This course was written to help you through every step of the learning process. We will use the widely adopted property graph model and the Cypher query language in our explanations, both of which are supported by Neo4j.

**What is a Graph?**

A graph is composed of two elements: a node and a relationship.

Each node represents an entity (a person, place, thing, category or other piece of data), and each relationship represents how two nodes are associated. For example, the two nodes *cake* and *dessert* would have the relationship *is a type of* pointing from *cake* to *dessert*.

This general-purpose structure allows you to model all kinds of scenarios: from a system of roads, to a network of devices, to a population’s medical history or anything else defined by relationships.

**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 On line 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.

**What is Neo4j**

* 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**

**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 database access.

Cypher is designed to be a human-friendly query language, suitable for both developers and any other professionals. Our guiding goal is to make the simple things easy, and the complex things possible. Optimized for being read by humans, Cypher’s construct uses English prose and iconography 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. 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 user, 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 keywords like WHERE and ORDER BY are inspired by SQL. Pattern matching borrows expression approaches from SPARQL. 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 as openCypher, 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.

**What is Cypher: Additional Resources**

This course gives you a basic understanding of Cypher. It’s a broad language with many cool features. When in doubt, please use the resources below to find out more:

* [Cypher Reference card](http://neo4j.com/docs/cypher-refcard/current/)
* [Cypher Documentation](http://neo4j.com/developer/cypher)
* [Cypher Reference Manual](http://neo4j.com/docs/developer-manual/current/#cypher-query-lang)
* [Graph Gist Challenges: Interactive Graph Models](http://neo4j.com/graphgists)
* [More Learning Resources](http://neo4j.com/developer/resources/)
* [openCypher Project](http://openCypher.org/)

**The Movie Database**

The data model in this tutorial includes nodes with three different labels (each with their own properties), and six different types of relationships (one of which has its own property). The underlying structure of the database is visualized in the image below.

In brief, the graph is made up of Person, Movie, and Genre nodes that are related to each other in various ways.

This tutorial will use the Cypher widget pre-populated with parts of the movie database. We use the model below to introduce and explain the Cypher concepts you need to know to build graph powered applications.

**A note on style**

Throughout this training, we will refer to nodes and property key/value pairs using monospace and real-life instances of the nodes using italics.

For example,

We set the title property of the Movie â€‹*The Matrix*â€‹ to "The Matrix"

**Patterns of Nodes and Relationships**

**Nodes**

Cypher uses a pair of parentheses (usually containing a text string) like (), (foo) to represent a node, i.e. an entity of your domain. This is reminiscent of a circle or a rectangle with rounded corners. Here are some ASCII-art representations of nodes, providing varying types and amounts of detail:

()

(matrix)

(:Movie)

(matrix:Movie)

(matrix:Movie {title: "The Matrix"})

(matrix:Movie {title: "The Matrix", released: 1999})

(matrix:Movie:Promoted)

The simplest form, (), represents an anonymous, uncharacterized node.

If we want to refer to the node elsewhere, we can add an variable: (matrix). Variables are restricted (ie, scoped) to a single statement; a variable may have different (or no) meaning in another statement.

The Movie **label** (prefixed in use with a colon) declares the node’s type or role. Note that nodes can have multiple labels. Labels are used to restrict search patterns, keeping them from matching structures without the labels in the query.

Neo4j’s indexes also use labels. Each index is specific to the combination of a label and a property.

The node’s properties (title, released, et cetera) are represented as a list of key/value pairs, enclosed within a pair of braces:

|  |
| --- |
| {title:"A Title", released:2000, …​} |

**Properties** can be used to store information and/or restrict patterns. For example, we could MATCH and RETURN nodes whose title is "The Matrix".

These attributes look similar to JSON structures:

MATCH (movie:Movie {title:"The Matrix"})

RETURN movie

| **movie** |
| --- |
| {"tagline":"Welcome to the Real World","title":"The Matrix","released":"1999"} |

**Labels**

Labels allow us to group our nodes. For example, we might want to distinguish movies from people or animals, both of whom act in films. By matching for (actor:Person)-[:ACTED\_IN]->(movie), Neo4j will return "Clint Eastwood", but not "Clyde"-his pet orangutan in *Every Which Way but Loose*.

Labels are usually used like this:

MATCH (node:Label) RETURN node

MATCH (node1:Label1)-[:REL\_TYPE]->(node2:Label2)

RETURN node1, node2

Although they are not required, Cypher uses labels to make better decisions on how to optimize your query, making them vital to success in Neo4j and Cypher.

**Relationships**

The missing piece of the Cypher snippets in the previous section is that they didn’t say anything about the relationship between the nodes, which add all the contextual information to our data. We want to be able to view a person in his or her relationship to Movie as an "Actor", "Director" and/or "Producer". In short, we need to be able to describe the types of relationships in our Cypher queries.

First and foremost, relationships are arrows pointing from one node to another, much like --> or <--. But we can add detail about them as needed within a pair of square brackets.

If we wanted to retrieve everyone who had acted in a movie, we would describe the pattern (actor:Person)-[:ACTED\_IN]->(movie:Movie) to retrieve only nodes that had a relationship type ACTED\_IN with other nodes.

Or generally:

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

Sometimes we need access to information about a relationship (e.g. its type or properties). For example, we might want to output the roles that an actor played in a movie.

As with nodes, we can use variables for relationships (in front of the :TYPE). If we tried to match (actor)-[rel:ACTED\_IN]->(movie) we would be able to output the rel.roles for each of the actors in all of the movies that they acted in.

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

RETURN rel.roles

Or generally:

MATCH (node1)-[rel:TYPE]->(node2)

RETURN rel.property

**Patterns**

Combining the syntax for nodes and relationships, we can express patterns. The following is a simple pattern (or fact) in this domain:

MATCH (matrix:Movie {title:"The Matrix"} )<-[role:ACTED\_IN {roles:["Neo"]}]-(keanu:Person {name:"Keanu Reeves"})

RETURN matrix, role, keanu

| **matrix** | **role** | **keanu** |
| --- | --- | --- |
| {"tagline":"Welcome to the Real World","title":"The Matrix","released":1999} | {"roles":["Neo"]} | {"born":1964,"name":"Keanu Reeves"} |

Like with node labels, the relationship type ACTED\_IN is added as a symbol, prefixed with a colon. For example: :ACTED\_IN. Variables (e.g., role) can be used elsewhere in the statement to refer to the relationship. Node and relationship properties use the same notation. In this case, we used an array property for the roles, allowing multiple roles to be specified.

To increase modularity and reduce repetition, Cypher allows patterns to be assigned to variables. This allows the matching paths to be inspected, used in other expressions, and more.

In the example below, the cast variable contains two nodes and the connecting relationship for each path that was found or created. There are a number of functions to access details of a path, including nodes(path), rels(path) (same as relationships(path)), and length(path).

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

RETURN cast

**Resources**

* [Graphs, Patterns, and Cypher](http://neo4j.com/docs/developer-manual/current/#cypher-getting-started)
* [Patterns in Practice](http://neo4j.com/docs/developer-manual/current/#introduction-pattern)

**Nodes and their Properties**

Let’s start by following along with the video and adding a node to the graph.

Run the following query, replacing **My Name** with your name in quotes. (If you happen to have the same name as a famous actor, you might want to change what name you put in):

CREATE (me:Person {name: "My Name"})

RETURN me

You will see the new node returned and also as part of the visualization. You can also easily check for its existence with the following query.

MATCH (me:Person)

WHERE me.name="My Name"

RETURN me.name

or in a short-hand syntax:

MATCH (me:Person {name:"My Name"})

RETURN me.name

**The All Nodes Query**

If we wanted to return **all nodes** in the graph, we can use the following query:

MATCH (n)

RETURN n

|  |  |
| --- | --- |
| Warning | In a larger graph this will return A LOT of data, usually you add a LIMIT 100 or such. |

The query is doing a full graph search. It visits every single node to see whether it matches the pattern of (n). In this concrete case, the pattern is simply a node that may or may not have a label or relationships, so it will match every single node in the graph. The RETURN clause then returns all of the information about each of those nodes, including all of their properties.

**Adding Properties**

To get set up, let’s add the movie Mystic River.

CREATE (movie:Movie {title: "Mystic River", released:1993})

Now, find the actor Kevin Bacon and the movie Mystic River and add the relationship between the movie and the actor to the dataset.

MATCH (kevin:Person) WHERE kevin.name = "Kevin Bacon"

MATCH (mystic:Movie) WHERE mystic.title = "Mystic River"

CREATE (kevin)-[r:ACTED\_IN {roles:["Sean"]}]->(mystic)

RETURN mystic,r, kevin

Let’s say we wanted to add a tagline to the Mystic River :Movie node we’ve just added. First, we have to locate the single movie again by its title, then SET the tagline property. The query:

MATCH (movie:Movie)

WHERE movie.title = "Mystic River"

SET movie.tagline = "We bury our sins here, Dave. We wash them clean."

RETURN movie.title AS title, movie.tagline AS tagline

| **title** | **tagline** |
| --- | --- |
| Mystic River | We bury our sins here, Dave. We wash them clean. |

Because Neo4j is schema-free, you can add any property you want to any node or relationship.

What if you want to update a property? Mystic River was actually released in 2003, not 1993.

We can fix that with the following query:

MATCH (movie:Movie)

WHERE movie.title = "Mystic River"

SET movie.released = 2003

RETURN movie.title AS title, movie.released AS released

| **title** | **released** |
| --- | --- |
| Mystic River | 2003 |

The syntax is the same for updating or adding a property. You SET a property. If the property exists, SET will update it. If the property doesn’t exist, SET will add it.

**Exercise: Updating a relationship property**

Let’s change the role of **Kevin Bacon** in **Mystic River** from ["Sean"] to ["Sean Devine"].

We should find the ACTED\_IN relationship between the Person and Movie using MATCH and then use SET to update the relationship property as we learned when creating the :Movie node.

**Nodes and Properties: Resources**

* [Create](http://neo4j.com/docs/developer-manual/current/#query-create)
* [Parameters](http://neo4j.com/docs/developer-manual/current/#cypher-parameters)
* [Setting all properties on a node](http://neo4j.com/docs/developer-manual/current/#_setting_all_properties_on_node)

**Relationships**

Adding a relationship is similar to adding a node, but we CREATE the relationship with the *relationship* syntax (n)-[:REL\_TYPE {prop: value}]->(m):

Let’s create ourselves first in this new database:

CREATE (me:Person {name:"My Name"})

RETURN me.name

And then let’s rate the movie Mystic River (or any other movie that you want to rate).

MATCH (me:Person), (movie:Movie)

WHERE me.name="My Name" AND movie.title="Mystic River"

CREATE (me)-[r:REVIEWED {rating:80, summary:"tragic character movie"}]->(movie)

RETURN me, r, movie

or

MATCH (me:Person {name:"My Name"}),(movie:Movie {title:"Mystic River"})

CREATE (me)-[r:REVIEWED {rating:80, summary:"tragic character movie"}]->(movie)

RETURN me, r, movie

**Exercise: Two Nodes, One Relationship**

Let’s say we wanted to return all of the nodes that have relationships to another node. This is still going to return every single node that has a relationship to another node, along with the other node. But it’s moving us in an important direction, so stay with us for a little longer.

To describe this query, we’d write:

MATCH (n)-->(m)

RETURN n, m;

This will return every pair of nodes with a relationship going from n to m.

**Exercise: Add Clint Eastwood as the director of Mystic River**

Can you add a director for the movie Mystic River? Clint Eastwood DIRECTED this movie.

If you want to make sure that only **one** relationship is created, no matter how often you run this statement, use MERGE instead. MERGE has **get-or-create** semantics. It tries to find the patterns you specify. If it finds them, it will return the data. Otherwise it creates the structure in the graph.

**Relationships: Resources**

* [Operators](http://neo4j.com/docs/developer-manual/current/#query-operators)
* [Create](http://neo4j.com/docs/developer-manual/current/#query-create)
* [Merge](http://neo4j.com/docs/developer-manual/current/#query-merge)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)
* [Set](http://neo4j.com/docs/developer-manual/current/#query-set)
* [Where](http://neo4j.com/docs/developer-manual/current/#query-where)

**Deleting Nodes**

Previously, we added ourselves to the graph. If you didn’t do that, add yourself now using the following query (replacing "My Name" with your name):

CREATE (me:Person {name:"My Name"});

Let’s then run the following query to make sure you have been added successfully to the graph.

MATCH (p:Person {name:"My Name"})

RETURN p.name

To remove both yourself and any relationships you may or may not have, you need to run:

MATCH (me:Person {name:"My Name"})

OPTIONAL MATCH (me)-[r]-()

DELETE me,r

It turns out there is another node in the graph that also needs to be deleted. Run the following query:

MATCH (matrix:Movie {title:"The Matrix"})<-[r:ACTED\_IN]-(actor)

RETURN actor.name, r.roles

It’s looking for and returning actors who played in the Matrix.

But wait, take a look at the results! Who is Emil? There is nobody (character or actor) named Emil in the Matrix. We need to delete this person.

Go ahead and delete Emil.

Did it work? No? Check out the next section.

**Deleting Nodes and Relationships**

This query statement will delete both the relationship and the node, even though there may be no relationships.

MATCH (emil:Person {name:"Emil Eifrem"})

OPTIONAL MATCH (emil)-[r]-()

DELETE emil,r

The first MATCH is obvious, it finds the node we’re looking for. The WHERE statement belongs to the first MATCH.

The second is an OPTIONAL MATCH. It tries to find nodes matching the pattern, if it doesn’t find anything it returns a single row with null values. But it will always return at least one row. You can also filter the optional match with a WHERE statement.

As this is a frequent task, DETACH DELETE was added to Cypher, which deletes a node with all its relationships.

**Deleting: Resources**

* [Create](http://neo4j.com/docs/developer-manual/current/#query-create)
* [Delete](http://neo4j.com/docs/developer-manual/current/#query-delete)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Optional Match](http://neo4j.com/docs/developer-manual/current/#query-optional-match)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)

**Lab: All Characters in the Matrix**

Using the syntax we’ve covered so far, RETURN a list of all the characters in the movie The Matrix.

**Hint**

* Movies have the label Movie and a title property you want to compare to.
* We’re looking for the characters—​the roles which are a property of the ACTED\_IN relationships—​not the names of the actors.

If you see all the usual suspects, you’re good.

**Order By, Limit, and Skip**

**ORDER BY**

In Cypher it’s easy to order results using an ORDER BY command. Let’s say we wanted to display the oldest people in our database. We could use the following query:

MATCH (person:Person)

RETURN person.name, person.born

ORDER BY person.born

The query returns every actor ordered by their year of birth, so it will display the oldest (smallest a.born) first.

**LIMIT and SKIP**

Cypher supports easy pagination of record sets. It uses LIMIT and SKIP statements to reduce the number of records returned and to allow for paging through the results.

If we wanted to display the **second page** of 10 actors and movies they played in, we might use the following query:

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

RETURN actor.name AS Actor, movie.title AS Movie

SKIP 10 LIMIT 10;

We could also just use LIMIT if we only want the top-n elements within the result.

**Exercise: Return the five oldest people in the database**

**Solution: Return the five oldest people in the database**

MATCH (person:Person)

RETURN person

ORDER BY person.born

LIMIT 5;

**Using DISTINCT**

Often you find yourself wanting to return only distinct results for a query. For example, let’s look at the list of the oldest actors. Initially, we might try the following:

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

RETURN actor

ORDER BY actor.born

LIMIT 5

But if any of the five oldest actors were in more than one movie, we’ll get them multiple times. So the query we really want to run is:

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

RETURN DISTINCT actor

ORDER BY actor.born

LIMIT 5

**ORDER BY, LIMIT, and SKIP: Resources**

* [Order by](http://neo4j.com/docs/developer-manual/current/#query-order)
* [Limit](http://neo4j.com/docs/developer-manual/current/#query-limit)
* [Skip](http://neo4j.com/docs/developer-manual/current/#query-skip)
* [Distinct](http://neo4j.com/docs/developer-manual/current/#aggregation-distinct)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)

**Predicates and Conditions**

**Using WHERE**

Cypher provides a number of mechanisms for reducing the number of matching patterns returned in a result set.

Let’s start with a simple query:

MATCH (person:Person)

WHERE person.name = "Tom Hanks"

RETURN person

This will look through all the nodes in the graph with a label of Person, and if one has the name "Tom Hanks", it will RETURN that node.

There is a shorter version of this query, which adds the Properties to filter by the MATCH clause.

MATCH (person:Person {name:"Tom Hanks"})

RETURN person

**Filtering using Comparisons**

We can also filter by comparing properties of different nodes. For example, we could RETURN all of the actors who acted with Tom Hanks and are older than him:

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

WHERE tom.name="Tom Hanks"

AND actor.born < tom.born

RETURN actor.name AS Name

Note that we didn’t bother to put (movie) in the middle-just (), as we don’t need to know anything about the movie in which they worked together.

We can even add a little math to the RETURN clause along with the *alias* AS diff to show us the difference in ages:

MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED\_IN]->(movie:Movie),

(movie)<-[:ACTED\_IN]-(actor:Person) // note how the line break doesn't break the query

WHERE actor.born < tom.born

RETURN DISTINCT actor.name AS Name, (tom.born - actor.born) AS diff

Math or more general expressions can be used almost everywhere in Cypher.

**Filtering using Patterns: A Few Examples**

How would we find all the actors who worked with Gene Hackman?

**Solution: Find all the actors who worked with Gene Hackman**

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

WHERE gene.name="Gene Hackman"

RETURN DISTINCT other

How do we filter those actors to only actor-directors?

So far, we used paths as part of a MATCH clause, but it is also possible to use paths as filter expressions in the WHERE clause. Here we check their existence with the exists function

**Solution: Filter to only actor-directors**

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

WHERE gene.name="Gene Hackman"

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

RETURN DISTINCT other

**Example: Gene Hackman and not Robin Williams**

Find actors who worked with Gene Hackman, but not when he was also working with Robin Williams in the same movie.

**Solution: Gene Hackman and not Robin Williams**

MATCH (gene:Person {name:"Gene Hackman"})-[:ACTED\_IN]->(movie:Movie),

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

(robin:Person {name:"Robin Williams"})

WHERE NOT exists( (robin)-[:ACTED\_IN]->(movie) )

RETURN DISTINCT other

**Filters: Resources**

* [Boolean and Mathematical Operators](http://neo4j.com/docs/developer-manual/current/#query-operators)
* [Create](http://neo4j.com/docs/developer-manual/current/#query-create)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)
* [Where](http://neo4j.com/docs/developer-manual/current/#query-where)

**Review: Exploring the Movie Database**

**Exercise: Tom Hank’s Filmography**

If you wanted to **Find all of the movies that *Tom Hanks* acted in**, how might you do that?

**Tom Hank’s Filmography after 2000**

What if you wanted to limit that to movies which were released after *2000*? Note that there is a released property on Movie.

**Solution: Tom Hank’s Filmography after 2000**

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

WHERE tom.name="Tom Hanks"

AND movie.released > 2000

RETURN movie.title AS `Movie Title`

**Exercise: Find all movies Keanu Reeves has acted in**

**Exercise: Find all movies in which Keanu Reeves played the role Neo**

**Hints:**

* You need an variable for the relationship.
* The ACTED\_IN relationship has a roles property (which is an array).
* The syntax for seeing whether an element is in an array is {element} IN r.roles.
* Generally check for the existence of the value of {expression} IN {collection}

**Solution: Find all movies in which Keanu Reeves played the role Neo**

MATCH (keanu:Person)-[r:ACTED\_IN]->(movie)

WHERE keanu.name="Keanu Reeves"

AND "Neo" IN r.roles

RETURN movie.title

**Lab Resources: Exploring the Movie Database**

* [Boolean and Mathematical Operators](http://neo4j.com/docs/developer-manual/current/#query-operators)
* [Collection functions](http://neo4j.com/docs/developer-manual/current/#query-functions-collection)
* [Create](http://neo4j.com/docs/developer-manual/current/#query-create)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)
* [Where](http://neo4j.com/docs/developer-manual/current/#query-where)

**Matching Paths**

**What is a Path?**

A **path** is a series of connected nodes and relationships. Paths can be matched by a pattern.

**What can we do with Paths?**

**Alternative Notations**

There are a number of different ways we could write the query we just examined:

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

RETURN actor.name, movie.title, director.name;

That’s nice, but especially for long paths the MATCH might not fit into a single path expression, so we can break it down into two separate segments using a comma:

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

(movie)<-[:DIRECTED]-(director:Person)

RETURN actor.name, movie.title, director.name;

In this form, we’re "taking a breath" with the comma, but we still want to return all the actors who acted in a movie together with the directors of those movies.

**This expresses the same query and will return the same record set.**

|  |  |
| --- | --- |
| Note | We have repeated the variable movie in both segments of the MATCH clause. This is critical. If we didn’t do this we’d get a very different record set as it is that shared variable that connects the two segments of the match clause. |

There is yet another way we could express the same query:

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

RETURN actor.name, movie.title, director.name;

Notice that the director element of the match clause is the other way round. However, the directionality (shown by the arrow) is still the same. So the following two snippets are identical as far as Cypher (and common sense) are concerned.

Identical Cypher snippets:

(movie)<-[:DIRECTED]-(director)

(director)-[:DIRECTED]->(movie)

**Variable Length Paths**

In Cypher, we can describe variable length paths using a star: \*

MATCH (node1)-[\*]-(node2)

|  |  |
| --- | --- |
| Warning | This unbounded path will potentially match many million or billion paths in medium size graphs. It also is important to use relationship-types and directions if possible. Always use paths with an upper limit or between two anchored nodes. |

* Relationships that traverse any depth are: (a)-[\*]->(b)
* Specific depth of relationships are represented like (a)-[\*depth]->(b) to find all paths exactly depth steps long.
* Relationships from one to four levels deep are represented like so: (a)-[\*1..4]->(b)
* Relationships of type KNOWS at 3 levels distance: (a)-[:KNOWS\*3]->(b)
* Relationships of type KNOWS or LIKES from 2 levels distance: (a)-[:KNOWS|:LIKES\*2..]->(b)

**Matching Paths: Resources**

* [Boolean and Mathematical Operators](http://neo4j.com/docs/developer-manual/current/#query-operators)
* [Pattern Variables](http://neo4j.com/docs/developer-manual/current/#_pattern_variables)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)

**Returning Paths**

In addition to being able to MATCH paths, we can name paths and RETURN them as part of the result:

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

RETURN p;

This will RETURN all of the nodes and relationships for each path, including all of their properties. That’s interesting, but can be too much data. We might use the nodes() function just to RETURN the nodes in the path:

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

RETURN nodes(p);

There is a similar function for relationships:

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

RETURN rels(p);

Note that only connected patterns can be used to create named paths. If you have two patterns in your MATCH clause with a comma between them, you’d have to RETURN the results as two distinct named paths:

MATCH p1=(actor:Person)-[:ACTED\_IN]->(movie:Movie), p2=(director:Person)-[:DIRECTED]->(movie)

RETURN p1, p2;

**Exercise: Directors acting in their movies**

We’ve already seen how to RETURN all of the actors and directors in all of the movies:

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

RETURN actor.name, movie.title, director.name;

How would you change this query to RETURN only the directors who acted in their own movies? Return people who both acted and directed in the same movie and display their name.

**Hint**

If you’re having trouble, what would happen if you replaced the (d) and d.name with an (a) and a.name? Does that work? Why? How could you simplify that query?

**Returning Paths: Resources**

* [Boolean and Mathematical Operators](http://neo4j.com/docs/developer-manual/current/#query-operators)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)
* [Where](http://neo4j.com/docs/developer-manual/current/#query-where)

**Indexing and Labels**

Unlike other databases, Neo4j doesn’t use indexes to speed up JOIN operations. However, they are useful for finding your starting points by value, textual prefix or range. You’ll create a **label specific index** as indexes are bound to a concrete label-property combination.

So if you want to be able to search efficiently for Movies based on their title, you might run the following Cypher command:

CREATE INDEX ON :Movie(title)

To remove the index, use the following Cypher command:

DROP INDEX ON :Movie(title)

**Exercise: How would you create an index for searching people by name?**

**Solution: How would you create an index for searching people by name?**

CREATE INDEX ON :Person(name);

You don’t need to do anything to your queries to use these indexes. Run the commands above, followed by:

MATCH (gene:Person)-[:ACTED\_IN]->(movie),(other:Person)-[:ACTED\_IN]->(movie)

WHERE gene.name="Gene Hackman"

RETURN DISTINCT other;

The lookup of Gene Hackman will now be much faster, although with a small test data set the difference may not be noticeable.

**Exercise: Using Indexes**

Try to use the query below once with and once without an index.

MATCH (gene:Person)-[:ACTED\_IN]->(movie),(other:Person)-[:ACTED\_IN]->(movie)

WHERE gene.name="Gene Hackman"

RETURN DISTINCT other;

**Indexing and Labels: Create a Label-Specific Index**

Create an index on nodes labeled Person, indexing by the property name:

CREATE INDEX ON :Person(name);

Create an index on nodes labeled Movie, indexing by the property title:

CREATE INDEX ON :Movie(title);

**Indexing and Labels: Anchor Pattern Nodes in the Graph**

Return movies featuring both Tom Hanks and Kevin Bacon

MATCH (tom:Person)-[:ACTED\_IN]->(movie:Movie),(kevin:Person)-[:ACTED\_IN]->(movie)

WHERE tom.name="Tom Hanks" AND kevin.name="Kevin Bacon"

RETURN DISTINCT movie;

You can anchor one or more nodes of your pattern in the graph, by constraining their properties to a single fitting node. Then the pattern matching works much faster as Cypher doesn’t have to scan the whole graph to apply the patterns. If there is an index, Cypher will automatically use it.

**Indexing and Labels: Resources**

* [Create](http://neo4j.com/docs/developer-manual/current/#query-create)
* [Constraints](http://neo4j.com/docs/developer-manual/current/#query-constraints)
* [Indexes](http://neo4j.com/docs/developer-manual/current/#query-schema-index)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)
* [Where](http://neo4j.com/docs/developer-manual/current/#query-where)

**Aggregation**

Cypher provides support for a number of aggregate functions

count(x) Count the number of occurrences

min(x) Get the lowest value

max(x) Get the highest value

avg(x) Get the average of a numeric value

sum(x) Sum up values

collect(x) Collect all the values into an collection

More on aggregate functions can be found in the [Neo4j Manual](http://neo4j.com/docs/developer-manual/current/#query-aggregation).

**Aggregation: Collect**

Let’s say we wanted to display all movie titles that an actor participated in. We could use the following query:

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

RETURN person.name, collect(movie.title);

For every Person who has acted in at least one movie, the query will RETURN their name and an array of strings containing the movie titles.

Let’s look closer at the graph and at Tom Hanks' movies.

**Exercise: Return the names of all the directors each actor has worked with**

**Solution: Return the names of all the directors each actor has worked with**

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

RETURN person.name, collect(director.name);

But if the actor worked several times with the same director, they would appear repeatedly. We can use DISTINCT here to collect only the distinct set of director names.

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

RETURN person.name, collect(DISTINCT director.name) as directors;

**Exercise: Return the count of movies in which each actor has acted**

**Solution: Return the count of movies in which each actor has acted**

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

RETURN actor.name, count(movie);

**Exercise: Return the count of movies in which an actor and director have jointly worked**

**Solution: Return the count of movies in which an actor and director have jointly worked**

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

RETURN actor.name, director.name, count(movie);

**Aggregation: Top n**

If you’re interested in the top n results, use a count aggregation. If we were interested in the top ten actors who acted in the most movies, the query would look like this:

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

RETURN person.name, count(movie)

ORDER BY count(movie) DESC

LIMIT 10;

**Aggregation: Resources**

* [Aggregation](http://neo4j.com/docs/developer-manual/current/#query-aggregation)
* [Boolean and Mathematical Operators](http://neo4j.com/docs/developer-manual/current/#query-operators)
* [Create](http://neo4j.com/docs/developer-manual/current/#query-create)
* [Getting Correct Results](http://neo4j.com/docs/developer-manual/current/#cypherdoc-getting-the-results-you-want)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)
* [Where](http://neo4j.com/docs/developer-manual/current/#query-where)

**Lab: Who are the five busiest actors?**

Try to come up with a query that will display the five busiest actors, i.e. the ones who have been in the most movies.

**Hint**

Use aggregation and ordering

**Solution: Who are the five busiest actors?**

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

RETURN actor.name, count(movie)

ORDER BY count(movie) DESC

LIMIT 5;

**Lab: Recommendation Engine**

**Exercise: Recommend three actors that Keanu Reeves should work with (but hasn’t).**

This is kind of a **friends-of-a-friend** query, with ACTS\_IN acting as a proxy for FRIEND. There are different approaches for the recommendation. This is just one of them.

**Solution: Recommendation Engine**

MATCH (keanu:Person {name:"Keanu Reeves"})-[:ACTED\_IN]->()<-[:ACTED\_IN]-(coworker:Person),

(coworker)-[:ACTED\_IN]->()<-[:ACTED\_IN]-(coworkerOfCoworker)

WHERE coworkerOfCoworker <> keanu AND NOT((keanu)-[:ACTED\_IN]->()<-[:ACTED\_IN]-(coworkerOfCoworker))

RETURN coworkerOfCoworker.name, count(coworkerOfCoworker)

ORDER BY count(coworkerOfCoworker) DESC

LIMIT 3;

**Importing Data**

Throughout this course, we have been assuming data already exists in a database or is small enough to enter manually. However, what if you wanted to explore an already existing external dataset? How would you import data from a spreadsheet or relational database?

If you want to import data from CSV, you will need to 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 or 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 existing graph structures.

As CSV files usually represent either node- or relationship-lists, you will run multiple passes to create nodes and relationships separately.

The movies.csv file (sample below) contains the data that will populate the Movie nodes.

id,title,country,year

1,Wall Street,USA,1987

2,The American President,USA,1995

3,The Shawshank Redemption,USA,1994

The following query CREATE s the Movie nodes using the data from movies.csv as properties.

LOAD CSV WITH HEADERS

FROM "http://neo4j.com/docs/stable/csv/intro/movies.csv"

AS line

CREATE (movie:Movie { id:line.id, title:line.title, released:toInt(line.year) });

The persons.csv file (sample below) holds the data that will populate the :Person nodes.

id,name

1,Charlie Sheen

2,Oliver Stone

3,Michael Douglas

4,Martin Sheen

5,Morgan Freeman

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. As we only have to set the name of a person upon creation, we use the ON CREATE feature.

LOAD CSV WITH HEADERS

FROM "http://neo4j.com/docs/stable/csv/intro/persons.csv"

AS line

MERGE (actor:Person { id:line.id })

ON CREATE SET actor.name=line.name;

The roles.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 via their key "propertyId", and makes 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 "http://neo4j.com/docs/stable/csv/intro/roles.csv"

AS line

MATCH (movie:Movie { id:line.movieId })

MATCH (person:Person { id: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.csv

title;released;actor;born;characters

Back to the Future;1985;Michael J. Fox;1961;Marty McFly

Back to the Future;1985;Christopher Lloyd;1938;Dr. Emmet Brown

LOAD CSV WITH HEADERS

FROM "http://neo4j.com/docs/stable/csv/intro/movie\_actor\_roles.csv"

AS line FIELDTERMINATOR ";"

MERGE (movie:Movie { title:line.title })

ON CREATE SET movie.released = toInt(line.released)

MERGE (actor:Person { name:line.actor })

ON CREATE SET actor.born = toInt(line.born)

MERGE (actor)-[r:ACTED\_IN]->(movie)

ON CREATE SET r.roles = split(line.characters,",")

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 10000 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.

**Importing Data: Resources**

* [Boolean and Mathematical Operators](http://neo4j.com/docs/developer-manual/current/#query-operators)
* [Create](http://neo4j.com/docs/developer-manual/current/#query-create)
* [Load CSV](http://neo4j.com/docs/developer-manual/current/#query-load-csv)
* [Importing CSV Files with Cypher](http://neo4j.com/docs/developer-manual/current/#cypherdoc-importing-csv-files-with-cypher)
* [Match](http://neo4j.com/docs/developer-manual/current/#query-match)
* [Merge](http://neo4j.com/docs/developer-manual/current/#query-merge)
* [Return](http://neo4j.com/docs/developer-manual/current/#query-return)
* [Set](http://neo4j.com/docs/developer-manual/current/#query-set)
* [Using Periodic Commit](http://neo4j.com/docs/developer-manual/current/#query-periodic-commit)
* [Where](http://neo4j.com/docs/developer-manual/current/#query-where)

**Community Resources and Next Steps**

**In-Person Training**

If you enjoyed the online version of our training and want to continue to learn about Cypher, check out our <https://neo4j.com/events/#/events?area=Local&type=Training> [in-person, hands-on training classes]. They are taught around the world by experienced instructors.

**Online Groups and Resources**

If you’d like to learn more about using Neo4j, start by going to [the Neo4j developer page](http://neo4j.com/developer). It provides a range of topics for learning more about Neo4j.

If you have specific questions or problems, please ask on [Stack Overflow](http://stackoverflow.com/questions/tagged/neo4j), where the most knowledgable people can help you quickly. Come to the [Neo4j Google Group](http://groups.google.com/group/neo4j) if you want to discuss graph modeling questions, the Neo4j ecosystem or product features. You can also join our [public Slack channel](http://neo4j.com/slack) to get quick answers.

If you’d like to learn more about the company behind Cypher, the product and the enterprise features and support packages that are provided, you should check out [neo4j.com](http://neo4j.com/product). There you can also find more information about use cases and solutions.

Follow Neo4j on Twitter [@Neo4j](http://twitter.com/neo4j) and find local [Neo4j meetups](http://neo4j.meetup.com/) to connect with other developers interested in graph databases.

**Links**

* [Download Neo4j](http://neo4j.com/download/) and develop your first Neo4j project
* Ask questions on [Stack Overflow](http://stackoverflow.com/questions/tagged/neo4j)
* Discuss ideas on [Neo4j’s Google Group](http://groups.google.com/group/neo4j)
* Discuss ideas on [Neo4j’s Slack](http://neo4j.com/slack)
* Read the [Neo4j Documentation](http://neo4j.com/docs/stable/)
* Use the [Neo4j Cypher Reference Card](http://neo4j.com/docs/stable/cypher-refcard)