

Getting More Out of Neo4j

Table of Contents

About this module	1
Cypher parameters.....	2
Using Cypher parameters.....	2
Exercise 12: Using Cypher parameters.....	4
Analyzing Cypher execution	5
Monitoring queries.....	8
Exercise 13: Analyzing and monitoring queries	10
Managing constraints and node keys	11
Ensuring that a property value for a node is unique	11
Ensuring that properties exist	12
Retrieving constraints defined for the graph.....	14
Dropping constraints	15
Creating node keys	15
Exercise 14: Managing constraints and node keys	17
Importing data	18
Importing normalized data using LOAD CSV.....	18
Importing denormalized data.....	23
Importing a large dataset.....	23
Exercise 15: Importing data	24
Managing indexes	25
Creating indexes.....	25
Retrieving indexes.....	27
Dropping indexes.....	27
Exercise 16: Managing indexes	28
Accessing Neo4j resources	29
Check your understanding.....	30
Question 1	30
Question 2	30
Question 3	30
Summary	31

About this module

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

Suppose you do not want to **hard code** certain values in your Cypher statements? You may want to use a variety values at run-time when you are testing your Cypher and you don't want to change the code every time. In addition, you may include your Cypher in an application where parameters are passed in to the Cypher statements before it executes.

Using Cypher parameters

In your Cypher statements, a parameter 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 substituted in the Cypher statement.

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:

```
:param actorName => 'Tom Hanks'
```

Here is the result of executing the :param command:

```
$ :param actorName => 'Tom Hanks'
{
  "actorName": "Tom Hanks"
}
```

See :help param for usage of the :param command.

Successfully set your parameters.

Notice here that `param` is simply an object in your session that holds name/value pairs.

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

```
$ MATCH (p:Person)-[:ACTED_IN]->(m:Movie) WHERE p.name = $actorName RETURN m.released, m.title 0...
```

Table

A Text

</> Code

m.released	m.title
2012	"Cloud Atlas"
2007	"Charlie Wilson's War"
2006	"The Da Vinci Code"
2004	"The Polar Express"
2000	"Cast Away"
1999	"The Green Mile"
1998	"You've Got Mail"
1996	"That Thing You Do"
1995	"Apollo 13"
1994	"Forrest Gump"
1993	"Sleepless in Seattle"
1992	"A League of Their Own"
1990	"Joe Versus the Volcano"

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:

```
$ MATCH (p:Person)-[:ACTED_IN]->(m:Movie) WHERE p.name = $actorName RETURN m.released, m.title 0...
```

Table

A Text

</>

m.released	m.title
2000	"Jerry Maguire"
1992	"A Few Good Men"
1986	"Top Gun"

You can also use the JSON-style syntax to set all of the parameters in your Neo4j Browser session. In this example we set two parameters for our session:

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

With the result:

```
$ :param {actorName: 'Tom Cruise', movieName: 'Top Gun'}
```

```
{  
  "actorName": "Tom Cruise",  
  "movieName": "Top Gun"  
}
```

See `:help param` for usage of the `:param` command.

Successfully set your parameters.

Note: You can use `.param` or `:params` to set all of your parameters. If you want to remove an existing parameter from your session, you do by 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`:

```
$ :params
```

```
{  
  "actorName": "Tom Cruise",  
  "movieName": "Top Gun"  
}
```

See `:help param` for usage of the `:param` command.

Exercise 12: Using Cypher parameters

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

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 executes 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.

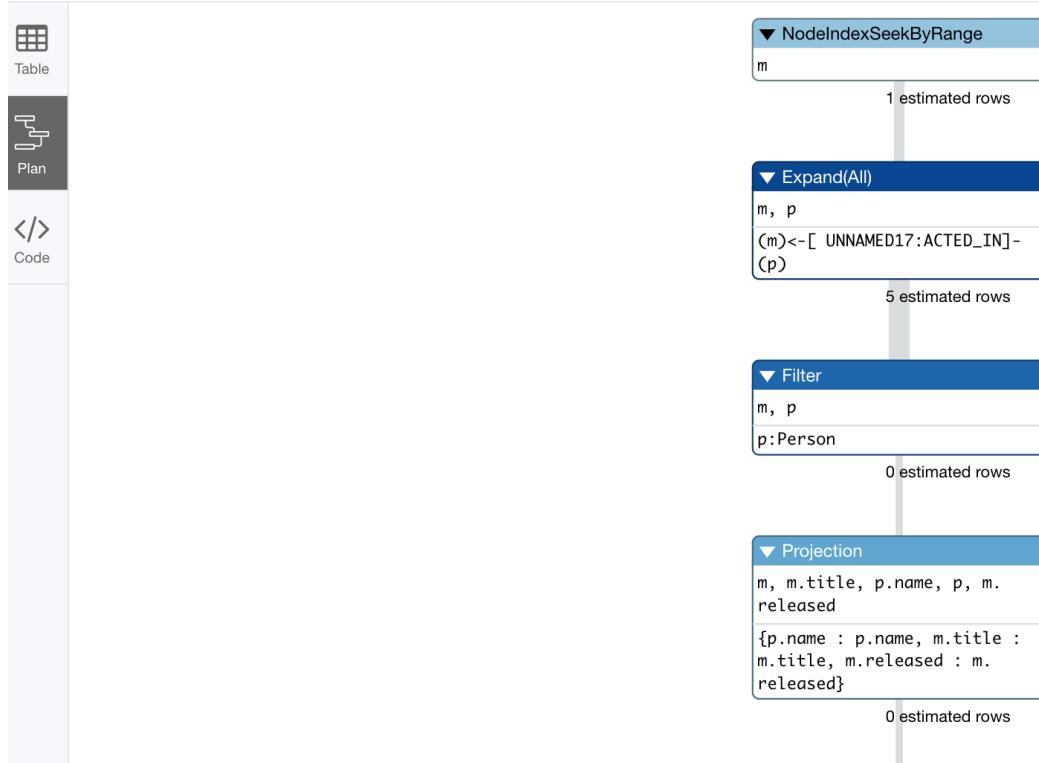
The **EXPLAIN** option will give you some information where you might be able to compare different Cypher statements where it tells you the stages of processing that will occur when the Cypher executes.

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
```

Here is the It returns

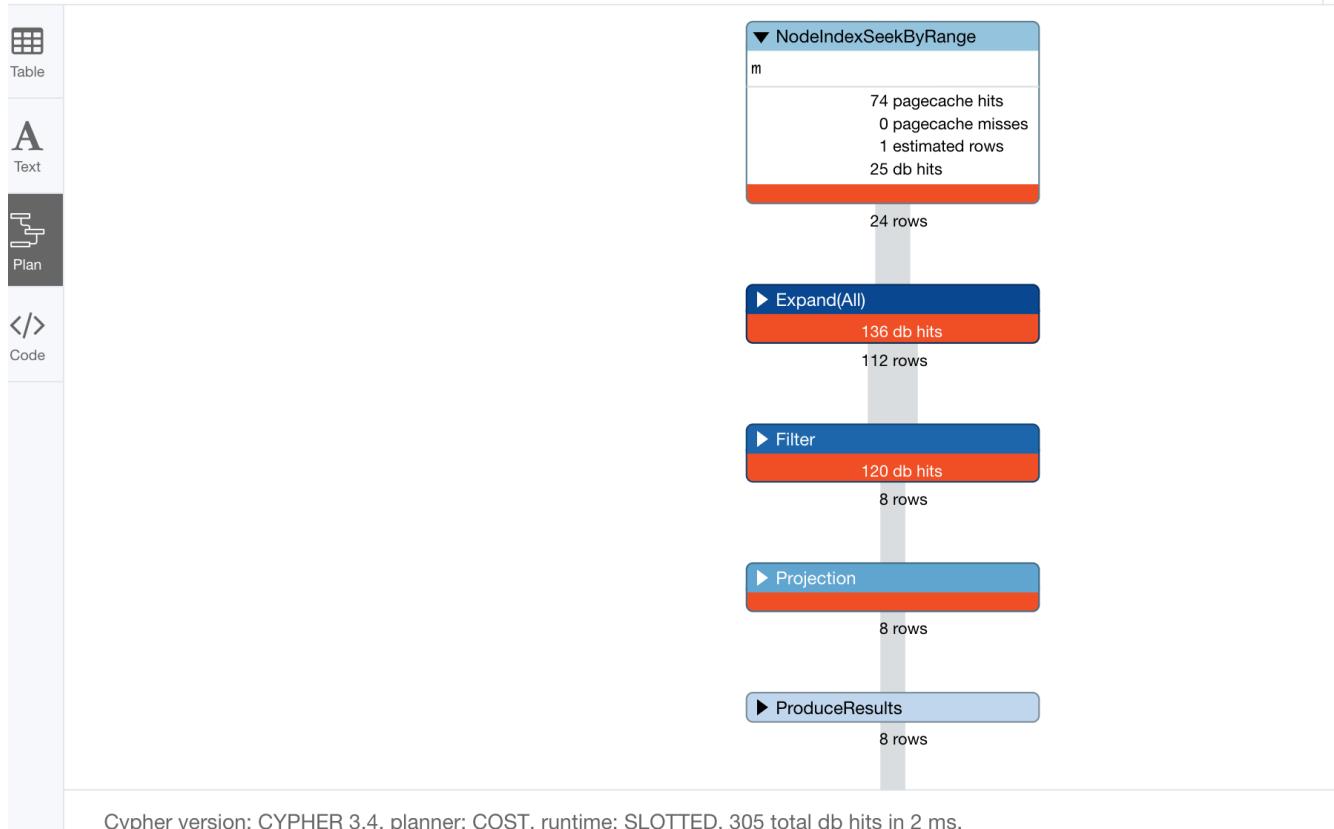
```
$ EXPLAIN MATCH (p:Person)-[:ACTED_IN]->(m:Movie) WHERE p.name = $actorName AND m.released < $year RE...
```



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. This Cypher statement ~~does not modify~~ the database, so 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 is the result returned using **PROFILE** for this Cypher statement:

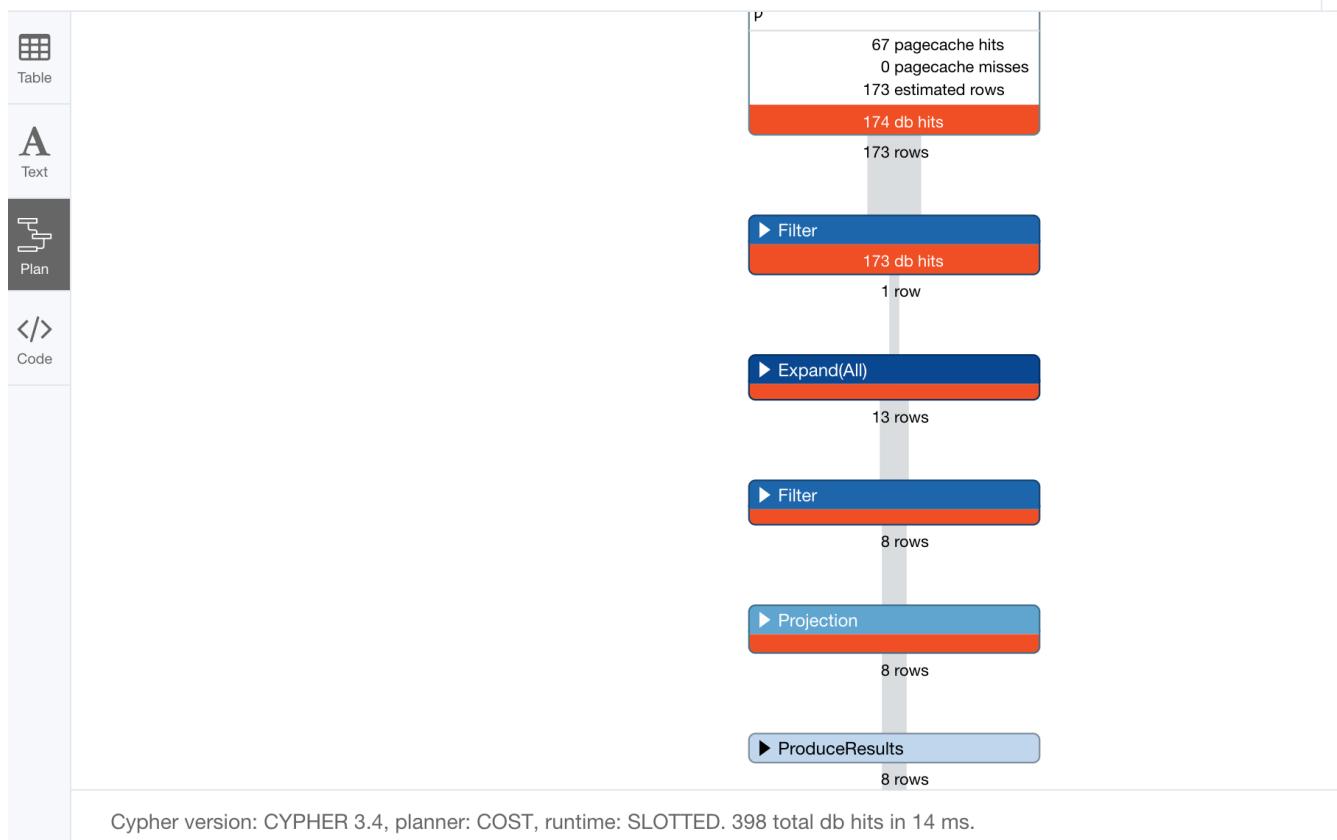
```
$ PROFILE MATCH (p:Person)-[:ACTED_IN]->(m:Movie) WHERE p.name = $actorName AND m.released < $year RE...
```



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:

```
$ PROFILE MATCH (p)-[:ACTED_IN]->(m) WHERE p.name = $actorName AND m.released < $year RETURN p.name, ...
```



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 `r` is 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:

\$:queries						
Database URI	User	Query	Params	Meta	Elapsed time	Kill
<code>bolt://localhost:7687</code>	neo4j	<code>CALL dbms.listQueries</code>	{}	{}	0 ms	
<code>bolt://localhost:7687</code>	neo4j	<code>match (a), (b), (c), (d), (e) return count (id(a))</code>	{}	{}	55526 ms	

The `:queries` command calls `dbms.listQueries` 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.

\$:queries						
Database URI	User	Query	Params	Meta	Elapsed time	Kill
<code>bolt://localhost:7687</code>	neo4j	<code>match (a), (b), (c), (d), (e) return count (id(a))</code>	{}	{}	135525 ms	
<code>bolt://localhost:7687</code>	neo4j	<code>CALL dbms.listQueries</code>	{}	{}	0 ms	

Here is what happens in the Neo4j Browser session where the long-running query was run:

The screenshot shows a Neo4j Browser session. In the top-left, there's a toolbar with icons for file operations like Open, Save, and Close. Below the toolbar, a dark sidebar on the left has a 'File' icon and the word 'Error'. The main area contains a red 'ERROR' button. Underneath it, the text 'Neo.TransientError.Transaction.Terminated' is displayed in bold black font. A gray callout box contains the error message: 'Neo.TransientError.Transaction.Terminated: The transaction has been terminated. Retry your operation in a new transaction, and you should see a successful result. Explicitly terminated by the user.' At the bottom of the main area, there's a red warning icon followed by the truncated text '⚠ Neo.TransientError.Transaction.Terminated: The transaction has been terminated. Retry your operation in a new transa...'. The top right of the browser window has standard window control buttons (minimize, maximize, close).

```
$ match (a), (b), (c), (d), (e) return count (id(a))
```

>Error

Neo.TransientError.Transaction.Terminated

Neo.TransientError.Transaction.Terminated: The transaction has been terminated. Retry your operation in a new transaction, and you should see a successful result. Explicitly terminated by the user.

⚠ Neo.TransientError.Transaction.Terminated: The transaction has been terminated. Retry your operation in a new transa...

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` <https://guides.neo4j.com/intro-neo4j-exercises> and follow the instructions for Exercise 13.

Managing constraints and node keys

You have learned how easy it is to create duplicate nodes in a graph. In most graphs, you will want to prevent duplication of data. One way that you can prevent duplication is by checking the existence of the exact node (with properties). This means that you must add additional clauses to test for existence when you create a node.

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 constraint that ensures that a value for a property is unique for all nodes of that type.
- Add a 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.

Important: Node keys are only supported in the Enterprise Edition of Neo4j and not available in the Community Edition.

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. In this situation, if a *Movie* node does not have a *title* property when the constraint is created, the creation of the constraint will be successful.

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

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



Added 1 constraint, completed after 370 ms.

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:

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

The screenshot shows the Neo4j browser interface. A red 'ERROR' button is visible. The error message is: **Neo.ClientError.Schema.ConstraintValidationFailed**. Below it, a detailed message reads: **Neo.ClientError.Schema.ConstraintValidationFailed: Node(874) already exists with label `Movie` and property `title` = 'The Matrix'**. At the bottom, there is a warning message: **⚠ Neo.ClientError.Schema.ConstraintValidationFailed: Node(874) already exists with label `Movie` ...**.

You will also see this error if you attempt to modify value of a property from a node where the uniqueness constraint has been created in the graph and the new value violates the assertion.

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:

```
$ MATCH (m:Movie) WHERE m.title STARTS WITH 'Something' RETURN m
```

The screenshot shows the Neo4j browser interface. On the left, there is a sidebar with three buttons: 'Graph' (selected), 'Table', and 'Text'. The main area displays a node labeled 'm' with the following properties:

```
{  
    "title": "Something's Gotta Give",  
    "released": 2003  
}
```

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)
```

Here is the result of running this Cypher statement:

The screenshot shows the Neo4j browser interface. On the left, there is a sidebar with a 'File' button (selected) and an 'Error' button. The main area shows an error message:

ERROR

Neo.DatabaseError.Schema.ConstraintCreationFailed

```
Neo.DatabaseError.Schema.ConstraintCreationFailed: Unable to create  
CONSTRAINT ON ( movie:Movie ) ASSERT exists(movie.tagline):  
Node(1208) with label `Movie` must have the property `tagline`
```

A red warning icon is present at the bottom left.

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. With the result:

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



Added 1 constraint, completed after 2 ms.

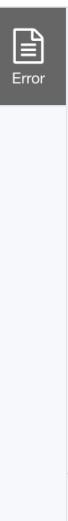
</>

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:

```
$ MATCH (p:Person), (m:Movie) WHERE p.name = 'Jessica Thompson' AND m.ti... | ↗ | ↘ | ^ | ○ | ×
```



ERROR

Neo.ClientError.Schema.ConstraintValidationFailed

```
Neo.ClientError.Schema.ConstraintValidationFailed: Relationship(1807) with
type `REVIEWED` must have the property `rating`
```

⚠ Neo.ClientError.Schema.ConstraintValidationFailed: Relationship(1807) 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

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:

```
$ CALL db.constraints
```



description

"CONSTRAINT ON (movie:Movie) ASSERT movie.title IS UNIQUE"

"CONSTRAINT ON ()-[reviewed:REVIEWED]-() ASSERT exists(reviewed.rating)"

Note: For any of the `CALL` `xx` statements in your Cypher, you can also specify them as `CALL db.xx()`. Adding the method notation 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)
```

With the result:

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



Removed 1 constraint, completed after 1 ms.

Creating node keys

A node key is used to define the uniqueness constraint for 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:

```
$ CREATE CONSTRAINT ON (p:Person) ASSERT (p.name, p.born) IS NODE KEY
```



ERROR

Neo.DatabaseError.Schema.ConstraintCreationFailed

```
Neo.DatabaseError.Schema.ConstraintCreationFailed: Unable to create CONSTRAINT  
ON ( person:Person ) ASSERT exists(person.name, person.born):  
Node(1183) with label `Person` must have the properties `name, born`
```

⚠ Neo.DatabaseError.Schema.ConstraintCreationFailed: Unable to create CONSTRAINT ON (person:Person) ASSERT exists..

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:

```
$ CREATE CONSTRAINT ON (p:Person) ASSERT (p.name, p.born) IS NODE KEY
```



Added 1 constraint, completed after 251 ms.

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})
```

Here is the result:

```
$ CREATE (:Person {name: 'Jessica Thompson', born: 0})
```



⚠ Neo.ClientError.Schema.ConstraintValidationFailed: Node(1223) already exists with label `Person` and properties `name` = '...

Exercise 14: Managing constraints and node keys

In the query edit pane of Neo4j Browser, execute the browser command: `:play https://guides.neo4j.com/intro-neo4j-exercises` and follow the instructions for Exercise 14.



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/>.

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 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 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 from each row read
create-or-merge-clause
```

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(*)
```

Here is the count result for this particular file:

```
$ LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/intro-neo4j/movies_to_load.csv" AS line RETURN count(*)
```

Table	count(*)
A	3

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
```

Here is the result of running the Cypher statement to visually inspect the data:

```
$ LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/intro-neo4j/movies_to_load.csv" AS line RETURN * LIMIT 1
```

line

```
{
    "summary": " Every dream has a
price.",
    "country": "USA",
    "id": "1",
    "title": "Wall Street",
    "year": "1987"
}
```

Started streaming 1 records after 245 ms and completed after 346 ms.

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 ~~left~~ 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), lTrim(line.summary)
```

Here we see how the data will be formatted before it is loaded:

```
$ LOAD CSV WITH HEADERS FROM 'http://data.neo4j.com/intro-neo4j/movies_to_load.csv' ...
```

line.id	line.title	toInteger(line.year)	lTrim(line.summary)
"1"	"Wall Street"	1987	"Every dream has a price."
"2"	"The American President"	1995	"Why can't the most powerful man in the world have the one thing he wants most?"
"3"	"The Shawshank Redemption"	1994	"Fear can hold you prisoner. Hope can set you free."

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: lTrim(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 `left` 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.

Here is the result of loading the `movies_to_load.csv` data into the graph:

```
$ LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/intro-neo4j/movies_to_load.csv" AS line CREATE (movie:Movie {...
```



Added 3 labels, created 3 nodes, set 12 properties, completed after 289 ms.

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 has 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.

Here is the result of loading the `persons_to_load.csv` data into the graph:

```
$ LOAD CSV WITH HEADERS FROM 'http://data.neo4j.com/intro-neo4j/persons_to_load.csv' AS li...
```



Added 5 labels, created 5 nodes, set 15 properties, completed after 359 ms.



Added 5 labels, created 5 nodes, set 15 properties, completed after 359 ms.

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)
```

Here is the result of loading the **roles_to_load.csv** data into the graph:

```
$ LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/intro-neo4j/roles_to_load.csv" AS line MATCH (movie:Movie { m...
```



Set 6 properties, created 6 relationships, completed after 323 ms.



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.

Here is the result of loading the **movie_actor_roles_to_load.csv** data into the graph:

```
$ LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/intro-neo4j/movie_actor_roles_to_load.csv" AS line FIELDTERMI...
```



Added 3 labels, created 3 nodes, set 9 properties, created 2 relationships, completed after 302 ms.

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 15: Importing data

In the query edit pane of Neo4j Browser, execute the browser command: `:play` <https://guides.neo4j.com/intro-neo4j-exercises> and follow the instructions for Exercise 15.

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 graph engine 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.

Here is a brief summary of when single-property and composite 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()

After you have weighed the performance gains for using indexes against the cost of graph space and extra overhead of index maintenance when the property is created or updated, you can create indexes in your graph. In most enterprises, the data model including indexes is owned by the architect of the application. 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.

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)
```

With the result:

```
$ CREATE INDEX ON :Movie(released)
```



Added 1 index, completed after 4 ms.

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']
```

With the result:

```
$ MATCH (m:Movie) WHERE m.released >= 2000 SET m.videoFormat = ['DVD', 'BlueRay']; MATCH (m:Movie) WH...
```

```
$ MATCH (m:Movie) WHERE m.released >= 2000 SET m.videoFormat = ['DVD', 'BlueRay']
```



```
$ MATCH (m:Movie) WHERE m.released < 2000 SET m.videoFormat = ['VHS', 'DVD']
```



Important: 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)
```

With the result:

```
$ CREATE INDEX ON :Movie(released, videoFormat)
```



Added 1 index, completed after 2 ms.

Retrieving indexes

Just as you can retrieve the constraints defined for the graph, you can retrieve the indexes:

```
CALL db.indexes
```

With the result:

\$ CALL db.indexes					
Table	description	label	properties	state	type
A	"INDEX ON :Movie(released)"	"Movie"	["released"]	"ONLINE"	"node_label_property" <pre>{ "version": "2.0", "key": "lucene+native" }</pre>
	"INDEX ON :Movie(released, videoFormat)"	"Movie"	["released", "videoFormat"]	"ONLINE"	"node_label_property" <pre>{ "version": "2.0", "key": "lucene+native" }</pre>
	"INDEX ON :Person(name, born)"	"Person"	["name", "born"]	"ONLINE"	"node_unique_property" <pre>{ "version": "2.0", "key": "lucene+native" }</pre>
	"INDEX ON :Movie(title)"	"Movie"	["title"]	"ONLINE"	"node_unique_property" <pre>{ "version": "2.0", "key": "lucene+native" }</pre>

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)
```

Here is the result:

```
$ DROP INDEX ON :Movie(released, videoFormat)
```



Removed 1 index, completed after 8 ms.

Exercise 16: Managing indexes

In the query edit pane of Neo4j Browser, execute the browser command: `:play` <https://guides.neo4j.com/intro-neo4j-exercises> and follow the instructions for Exercise 16.

Accessing Neo4j resources

There are many ways that you can learn more about Neo4j. A good starting point for learning about the resources available to you is the **Neo4j Learning Resources** page at <https://neo4j.com/developer/resources/>.

Check your understanding

Question 1

What Cypher keyword can you use to prefix any Cypher statement to examine how many db hits occurred when the statement executed?

Select the correct answer.

- ANALYZE
- EXPLAIN
- PROFILE
- MONITOR

Question 2

What types of constraints can you define for a graph that are asserted when a node or relationship is created or updated?

Select the correct answers.

- unique values for a property of a node
- unique values for a property of a relationship
- a node must have a certain set of properties with values
- a relationship must have a certain set of properties with values

Question 3

In general, what is the maximum number of nodes or relationships that you can create using LOAD CSV?

Select the correct answer.

- 1K
- 10K
- 1M
- 10M

Summary

You should now 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.