# Importing CSV Data into Neo4j course notes and links-

# Module 1: Importing Data

## **Module Overview**

In this module, you will learn about:

* The process for importing CSV data into Neo4j
* The CSV file structure
* Using LOAD CSV in Cypher to read a CSV file
* The Neo4j Sandbox, where you will import your data

## References

* [**Neo4j Sandbox**](https://sandbox.neo4j.com/)
* [**Neo4j Aura**](https://neo4j.com/cloud/platform/aura-graph-database/)
* [**LOAD CSV**](https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/)
* [**Neo4j documentation - Reading CSV Files**](https://neo4j.com/docs/getting-started/data-import/csv-import/#_reading_csv_files)
* [**Kaggle datasets**](https://www.kaggle.com/datasets)

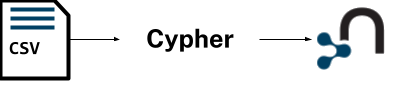
# Getting Started

Welcome to "Importing CSV Data using Cypher".

In this course, you will create a graph database of movies from a set of CSV files.



You will build an import process using Cypher, to create the nodes, relationships, and properties.



You don’t need anything other than a browser to access and complete this course.

A blank Neo4j Sandbox instance has been created for you to use during this course.

You can open a Neo4j Browser window throughout this course by clicking the [**Toggle Sandbox**](https://graphacademy.neo4j.com/courses/importing-cypher/1-importing-data/1-getting-started/) button in the bottom right-hand corner of the screen.

## What is Neo4j Sandbox?

Neo4j Sandbox is a free service that allows you to create pre-populated Neo4j instances.

Neo4j Sandbox is the perfect environment for experimenting with Neo4j.

You can log into Neo4j Sandbox and create a database with many pre-populated datasets by visiting [**sandbox.neo4j.com**](https://sandbox.neo4j.com/).

# The Data Import Process

Youtube video: <https://youtu.be/a0L5iQiIO80>

Transcript-

In this lesson, you will learn the process of importing CSV data using Cypher.

## **Source data**

When you import data into Neo4j, you typically start with a set of source files.

You may have exported this source data from:

* Relational databases
* Web APIs
* Public data directories
* BI tools
* Speadsheets (e.g. Excel or Google Sheets)

**EXPORT TO CSV**

Most data systems will have an option for exporting data to CSV files as it is a common format for data exchange.

The data in the source files may not be in the format needed for your graph data model:

* The source files could contain more data than you need.
* There may not be a 1-1 mapping of the data in a CSV file to a node or relationship.
* The data types might not map directly onto those supported in Neo4j.

Likely, you will need to transform the data before or during the import.

### **Create a graph**

Before you start the import process, you should:

1. Understand the data in the source CSV files.
2. Inspect and clean (if necessary) the data in the source data files.
3. Understand the graph data model you will be implementing during the import.

Before you import data into Neo4j, there is no data structure, only the database itself - as you import data, you create the graph data model.

Once you have the source data and a graph data model, you can create the graph by importing the data.

**DATA MAPPING**

You may find it useful to create a mapping document that shows how the source data maps to the graph data model.

The import involves creating Cypher code to:

* Read the source data.
* Transform the data as needed.
* Create nodes, relationships, and properties to create the graph.

Creating an import process will likely require multiple iterations as you build, test, and refactor.

# CSV files

CSV files store data in a structure where a special character, usually a comma, separates each value in a row, and each record is a new row in the file.

**csv**

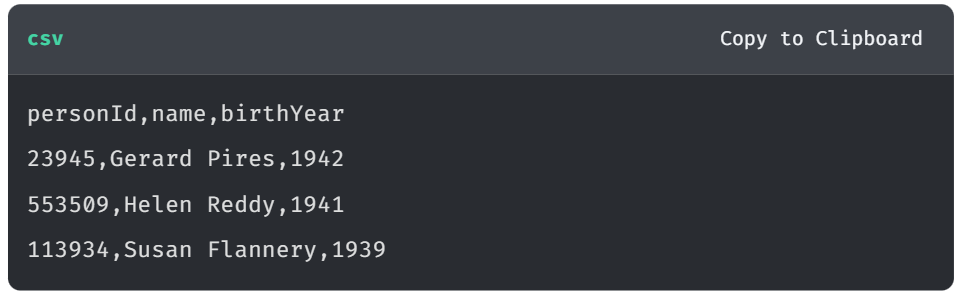
Copy to Clipboard

personId,name,birthYear

23945,Gerard Pires,1942

553509,Helen Reddy,1941

113934,Susan Flannery,1939



**VIEWING CSV FILES**

CSV files are simple text files - you can open them with any text editor (e.g. Notepad or TextEdit). You can also open them with a spreadsheet application (e.g. Excel or Google Sheets).

## Field Terminator

In the example above, a comma (,) separates each field, but other characters can be used, such as a tab (\t) or a pipe (|).

This is the same data as above but separated by a pipe character:

**csv**

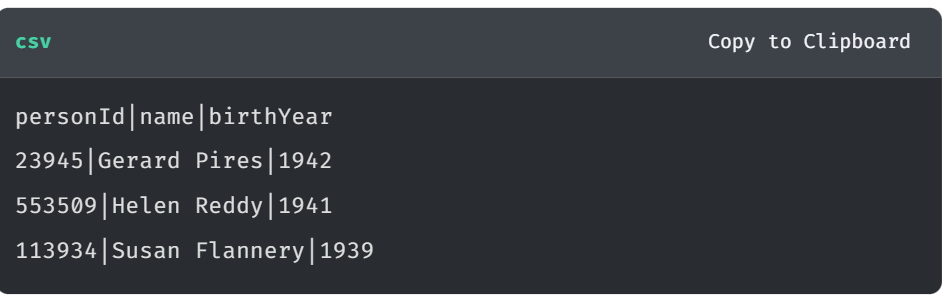
Copy to Clipboard

personId|name|birthYear

23945|Gerard Pires|1942

553509|Helen Reddy|1941

113934|Susan Flannery|1939



The character that separates the fields is called the **Field Terminator** or **Delimiter**.

## Headers

Typically the first row in a CSV file is a header row, which contains the names of the columns, although it is not required.

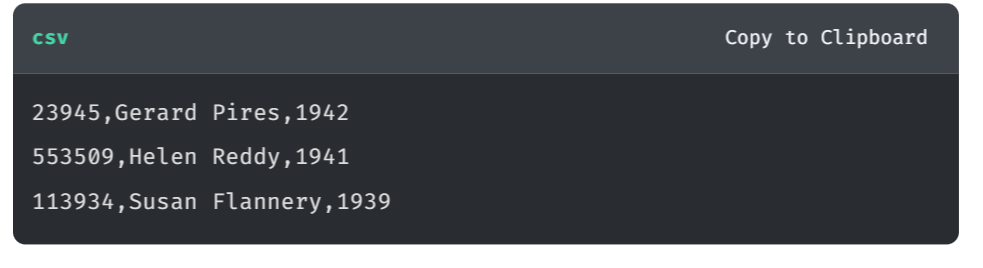
**csv**

Copy to Clipboard

23945,Gerard Pires,1942

553509,Helen Reddy,1941

113934,Susan Flannery,1939



When working with CSV files with no header row, you will need to know the order of the columns and refer to them by index.

Adding headers to CSV files is a good practice. It makes the data easier to understand and work with.

## Quotes

If the data contains the field terminator character, the data must be in quotes. e.g. if the names in the example were in the format last, first the data would need to be formatted as "last, first".

**csv**

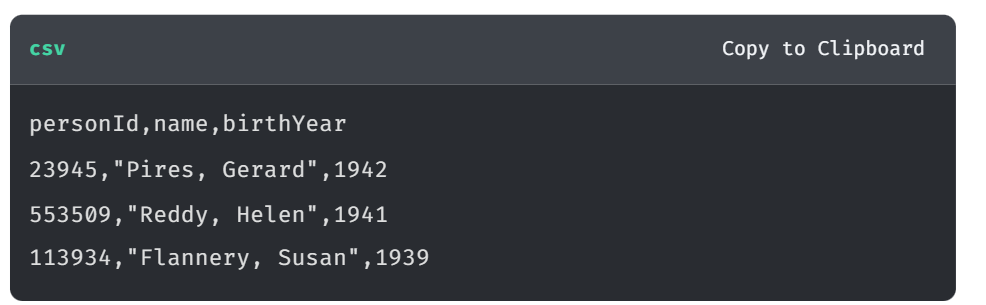
Copy to Clipboard

personId,name,birthYear

23945,"Pires, Gerard",1942

553509,"Reddy, Helen",1941

113934,"Flannery, Susan",1939



Most data applications will automatically quote data if it contains the field terminator character.

## Normalized data

If the source data is normalized (e.g. when exported from a relational data model) there will typically be multiple CSV files.

Each CSV file will represent a table in the relational data model, and the files will be related to each other by unique IDs.

In this normalized data example, there are three files for people, movies, and roles:

person.csv

**csv**

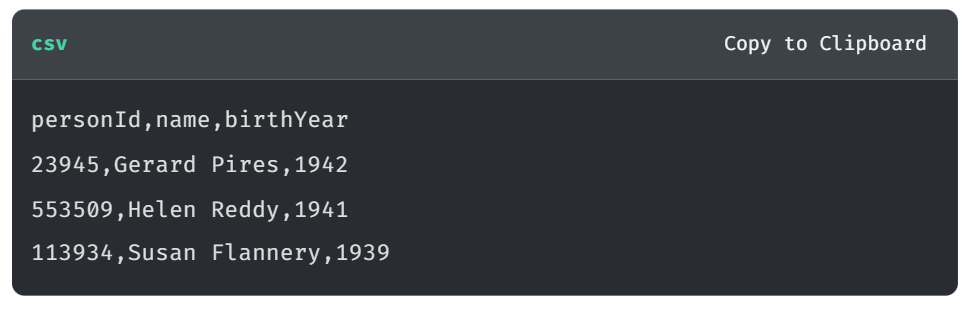
Copy to Clipboard

personId,name,birthYear

23945,Gerard Pires,1942

553509,Helen Reddy,1941

113934,Susan Flannery,1939



movies.csv

**csv**

Copy to Clipboard

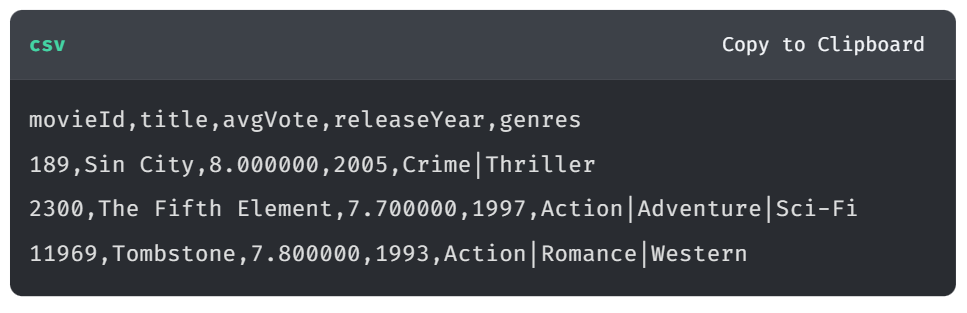
movieId,title,avgVote,releaseYear,genres

189,Sin City,8.000000,2005,Crime|Thriller

2300,The Fifth Element,7.700000,1997,Action|Adventure|Sci-Fi

11969,Tombstone,7.800000,1993,Action|Romance|Western

The genres field is a list of genres separated by a pipe (|) character.



roles.csv

**csv**

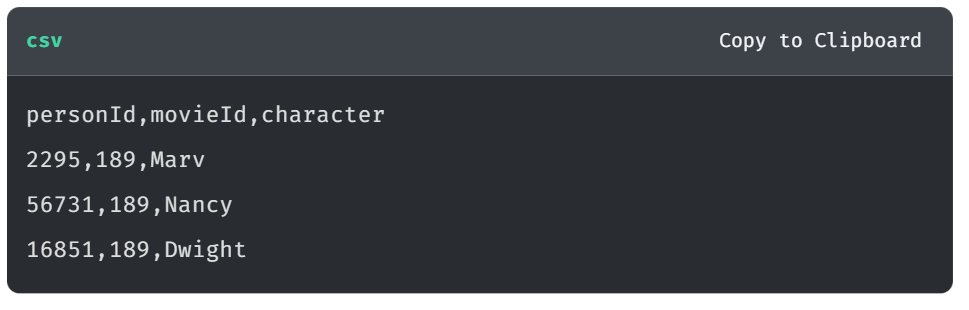
Copy to Clipboard

personId,movieId,character

2295,189,Marv

56731,189,Nancy

16851,189,Dwight



Notice that the **person.csv** file has a unique ID for every person, and the **movies.csv** file has a unique ID for every movie.

The **roles.csv** file relates a person to a movie and provides the characters.

## De-normalized data

If the source data is de-normalized, there will typically be a single CSV file. The single file will contain all the data, often duplicated where there are relationships between entities.

Here is an example of de-normalized data for the person, movie, and role data:

movies-n.csv

**csv**

Copy to Clipboard

movieId,title,avgVote,releaseYear,genres,personType,name,birthYear,character

2300,The Fifth Element,7.700000,1997,Action|Adventure|Sci-Fi,ACTOR,Bruce Willis,1955,Korben Dallas

2300,The Fifth Element,7.700000,1997,Action|Adventure|Sci-Fi,ACTOR,Gary Oldman,1958,Jean-Baptiste Emanuel Zorg

2300,The Fifth Element,7.700000,1997,Action|Adventure|Sci-Fi,ACTOR,Ian Holm,1931,Father Vito Cornelius

11969,Tombstone,7.800000,1993,Action|Romance|Western,ACTOR,Kurt Russell,1951,Wyatt Earp

11969,Tombstone,7.800000,1993,Action|Romance|Western,ACTOR,Val Kilmer,1959,Doc Holliday

11969,Tombstone,7.800000,1993,Action|Romance|Western,ACTOR,Sam Elliott,1944,Virgil Earp



De-normalized data typically represents data from multiple tables.

For example, the movie and person data (including the ID) is repeated in multiple rows in the file. A row represents a particular actor’s role in a movie. That is, for each role an actor played, the movie and person data will be duplicated.

How your data is structured will determine how you need to process it.

# Loading CSV files

Youtube video: <https://youtu.be/kVJy3nBrn6g>

Transcript-

In this lesson, you’ll learn how to use the [LOAD CSV](https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/) Cypher clause, including:

* The [LOAD CSV](https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/) syntax
* How to load headers within a CSV file
* Options for accessing a file

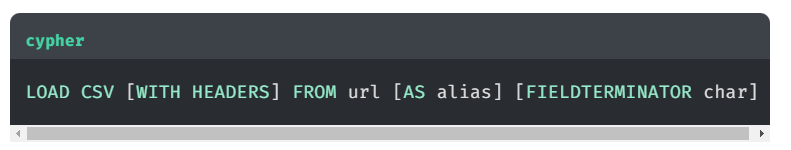
## **LOAD CSV**

The LOAD CSV Cypher clause reads data from a CSV file and returns the rows in the file.

LOAD CSV has the following syntax:

**cypher**

LOAD CSV [WITH HEADERS] FROM url [AS alias] [FIELDTERMINATOR char]



You can specify whether the file has a header row, the file location and the field terminator

You are going to load a [**CSV file that contains people data**](https://data.neo4j.com/importing-cypher/people.csv):

**csv**

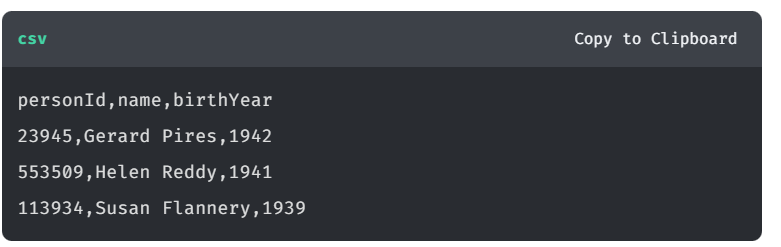
Copy to Clipboard

personId,name,birthYear

23945,Gerard Pires,1942

553509,Helen Reddy,1941

113934,Susan Flannery,1939



In the Sandbox run the following Cypher statement to load the people.csv file:

**cypher**

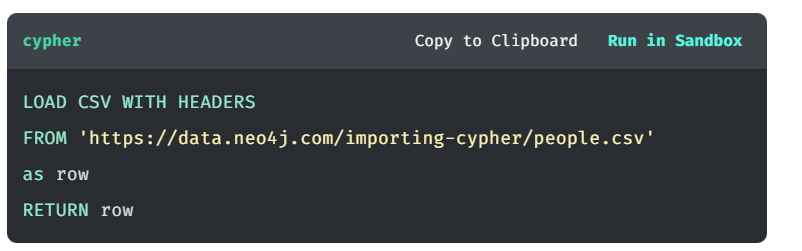
Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/people.csv'

as row

RETURN row



Review the data returned. Note the number of rows returned and the header names match those in the CSV file.

**json**

Copy to Clipboard

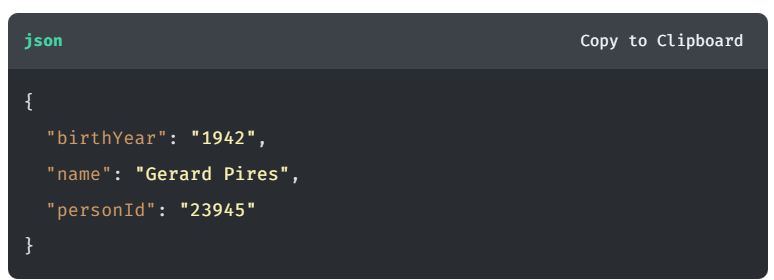
{

"birthYear": "1942",

"name": "Gerard Pires",

"personId": "23945"

}



The FIELDTERMINATOR wasn’t specified in the LOAD CSV clause because the default value is a comma. If the field terminator was a pipe character, the LOAD CSV clause would need to include the field terminator:

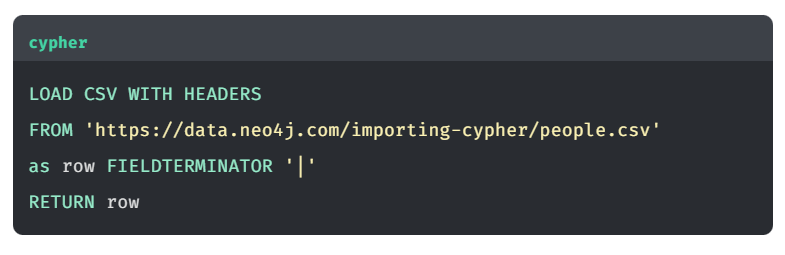
**cypher**

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/people.csv'

as row FIELDTERMINATOR '|'

RETURN row



## **Counting rows**

A simple check you can do to ensure all rows are loaded is to count the number of rows in the CSV file and compare it to the number of rows returned by the LOAD CSV clause.

You can return the number of rows in the CSV file using the COUNT function.

**cypher**

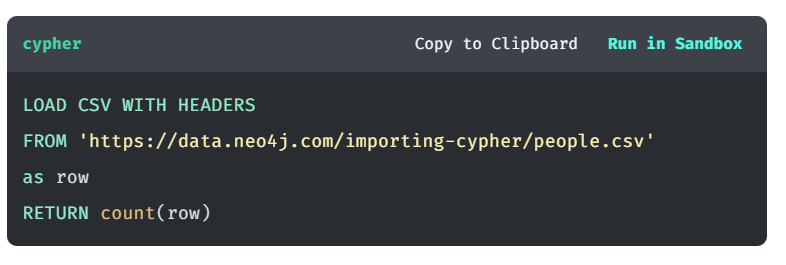
Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/people.csv'

as row

RETURN count(row)



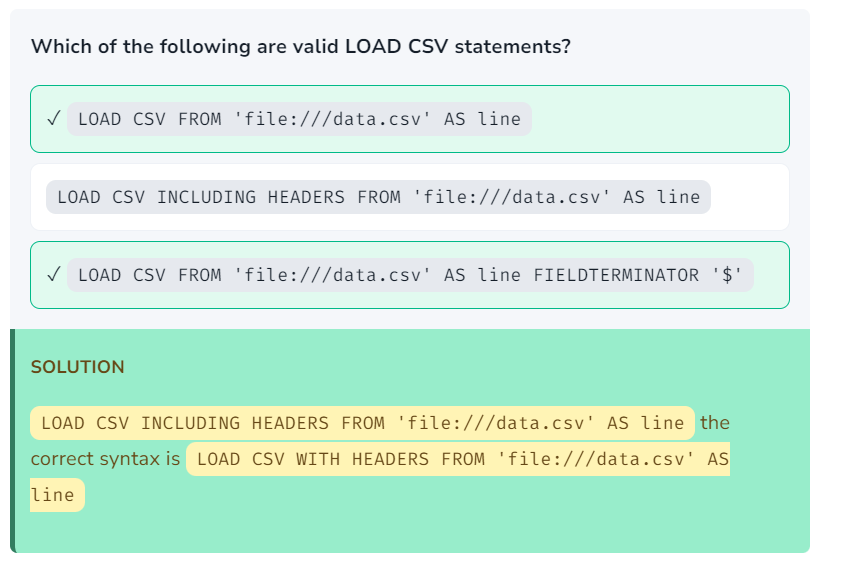
## **Accessing files**

LOAD CSV can access files on a Neo4j server’s file system or a remote machine.

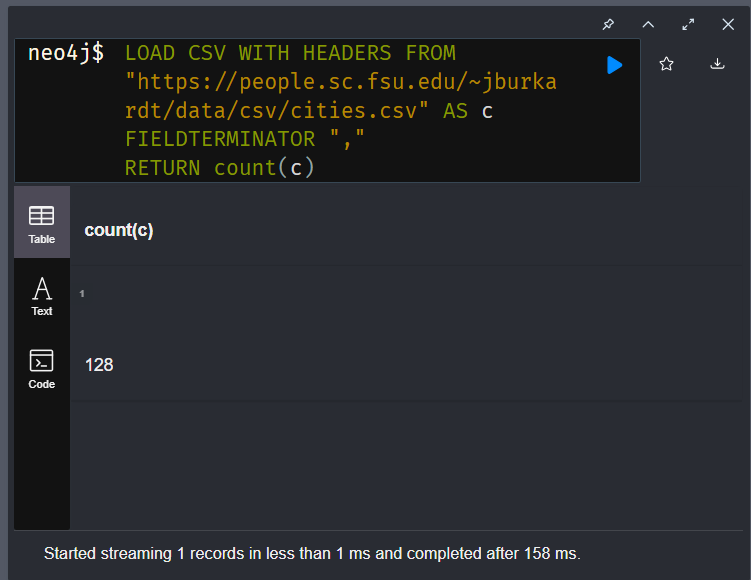
The Sandbox in this course and [**Aura DB**](https://neo4j.com/cloud/platform/aura-graph-database/) cloud service only allow access to remote files.

You can find more information in the [**Reading CSV Files**](https://neo4j.com/docs/getting-started/data-import/csv-import/#_reading_csv_files) section of the Neo4j documentation.

## Example Question-



Another Example-



# Module 2: Creating Nodes, Properties, and Relationships

## **Module Overview**

In this module, you will learn about:

* Creating nodes and relationships with data from a CSV file
* Assigning properties to nodes and relationships
* The importance of unique identifiers and how to create constraints

You will load data about movies and people related to those movies, including Person and Movie nodes, relationships between them, and constraints to ensure the data is unique.

## References

* [**Neo4j Sandbox**](https://sandbox.neo4j.com/)
* [**MERGE**](https://neo4j.com/docs/cypher-manual/current/clauses/merge/)
* [**LOAD CSV**](https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/)
* [**Constraints**](https://neo4j.com/docs/cypher-manual/current/constraints/)

# Nodes

Youtube video: <https://youtu.be/64VAmqT9XkY>

Transcript-

In this lesson, you will learn how to use the MERGE clause to create nodes using data from a CSV file.

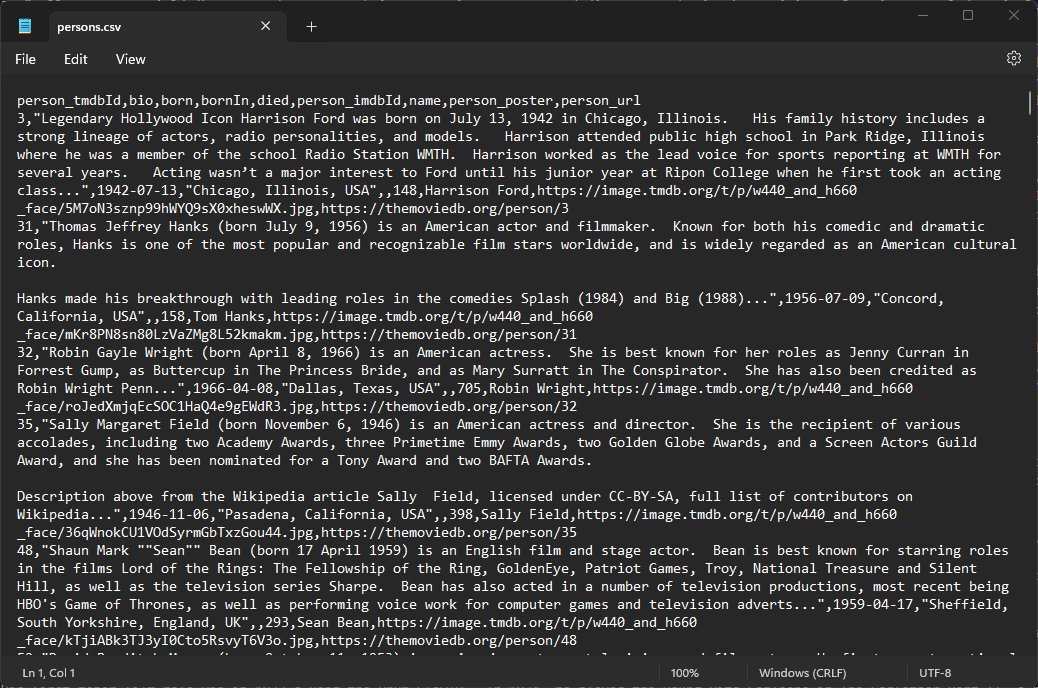
## Load the CSV file

You will load a CSV file of "person" data into Person nodes in Neo4j. The CSV file contains the following fields:

* person\_tmdbId
* bio
* born
* bornIn
* died
* person\_imdb
* Id
* name
* person\_poster
* person\_url

Follow these instructions to open the file and inspect the contents:

1. Download the [**persons.csv**](https://data.neo4j.com/importing-cypher/persons.csv) file
2. Open the file using a text editor to look at the contents.



You should see that the file contains headers, and the field delimiter is a comma (,).

1. Run the following Cypher statement to load the CSV file and return the contents:

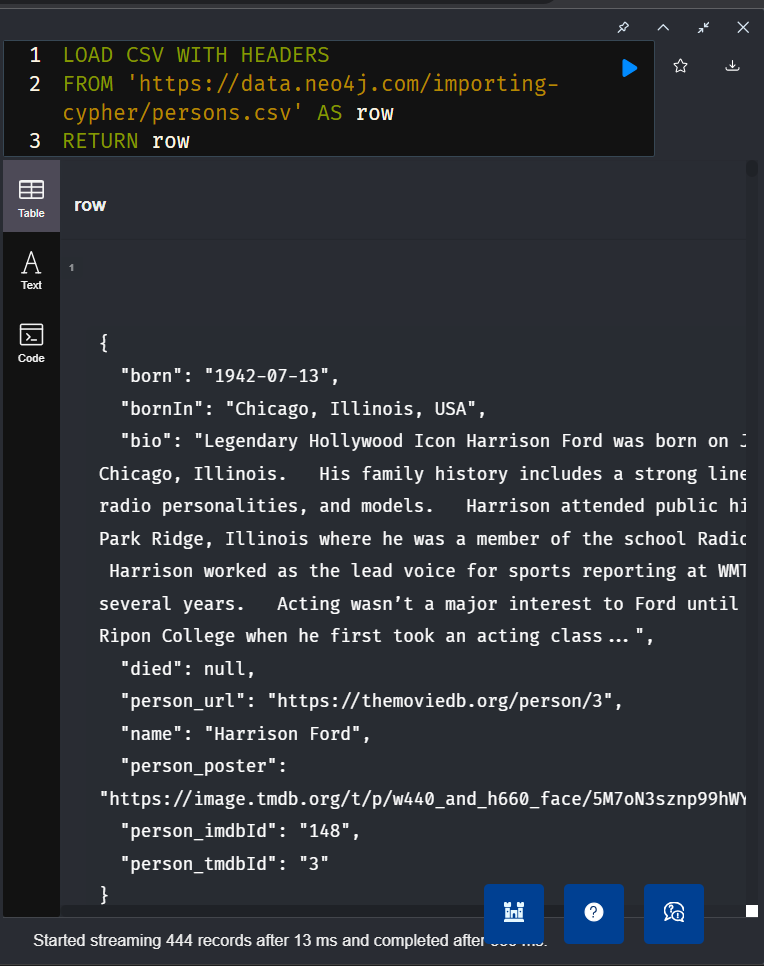
**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/persons.csv' AS row

RETURN row



The next step is to use the data in the CSV file to create Person nodes.

Before running it, review the following Cypher statement:

**cypher**

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/persons.csv' AS row

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId),

p.bornIn = row.bornIn,

p.name = row.name,

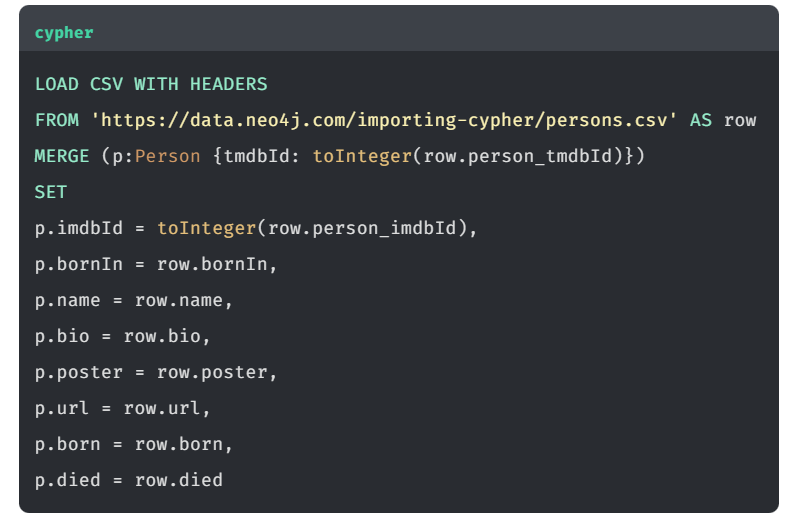
p.bio = row.bio,

p.poster = row.poster,

p.url = row.url,

p.born = row.born,

p.died = row.died



Try to answer the following questions:

1. Where does the CSV data come from?
2. What does the MERGE clause do?
3. What variable holds the data from the CSV file?
4. Where are the properties set?
5. Why is the toInteger function used?

Review the answers

1. The LOAD CSV clause loads the CSV file from the specified URL.
2. The MERGE clause creates a new Person if one does not already exist with the same tmdbId value.
3. The row variable holds the data from the CSV file.
4. The SET clause sets the properties of the Person node to the values of the corresponding fields in the CSV file.
5. The toInteger function converts the person\_tmdbId and person\_imdbId values from strings to integers.

## Create Person nodes

1. Run the Cypher statement to create the Person nodes:

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/persons.csv' AS row

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId),

p.bornIn = row.bornIn,

p.name = row.name,

p.bio = row.bio,

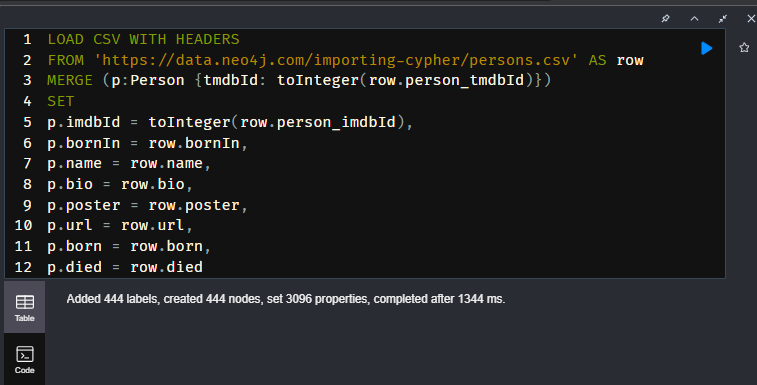
p.poster = row.poster,

p.url = row.url,

p.born = row.born,

p.died = row.died

The import should create 444 Person nodes.

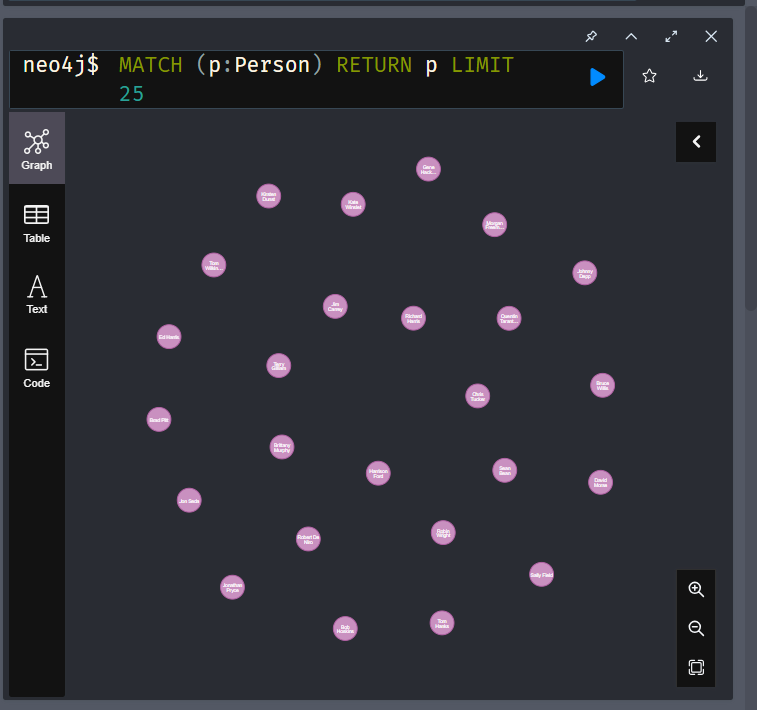


1. Confirm the data is in the graph by returning the first 25 Person nodes:

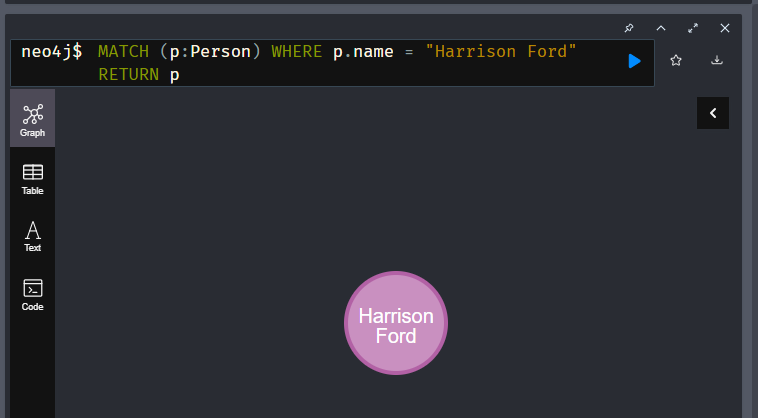
**cypher**

Copy to ClipboardRun in Sandbox

MATCH (p:Person) RETURN p LIMIT 25



1. Check the results. Do the nodes have the correct properties?



# Unique IDs and Constraints

A Neo4j best practice is to use an ID as a unique property value for each node.

Unique IDs help ensure duplicate data is not created. When you load data from CSV files, you rely heavily upon the IDs specified in the file. If the IDs in your CSV file are not unique for the same entity (node), you could create duplicate data. You may also have problems loading the data and creating relationships between nodes.

You can add constraints to your database to stop the creation of nodes with duplicate IDs.

## **Create a unique constraint**

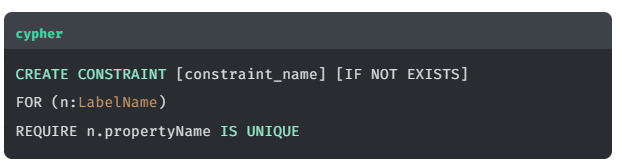
The syntax for creating a unique constraint on a property is:

**cypher**

CREATE CONSTRAINT [constraint\_name] [IF NOT EXISTS]

FOR (n:LabelName)

REQUIRE n.propertyName IS UNIQUE



The constraint is for a property on all nodes with a specified label.

The constraint\_name is optional, but it is good practice to specify one. If you do not specify a constraint name, Neo4j will create one for you.

The IF NOT EXISTS clause is also optional - if not used Neo4j will generate an error if the constraint already exists.

## **Person node constraint**

The Person nodes you created should all have a unique tmbdId property.

You can create a constraint for the tmdbId property to ensure that all Person nodes have a unique tmdbId property value.

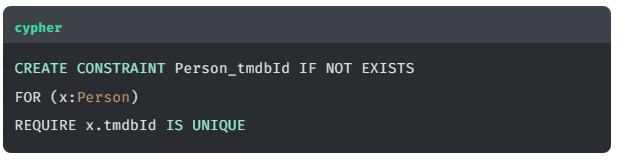
Review the following Cypher statement.

**cypher**

CREATE CONSTRAINT Person\_tmdbId IF NOT EXISTS

FOR (x:Person)

REQUIRE x.tmdbId IS UNIQUE



You should note that:

* The constraint name is Person\_tmdbId.
* The optional clause IF NOT EXISTS is used - without which Neo4j would raise an error if the constraint exists.
* It applies to all nodes with the Person label.
* It requires the tmdbId property to be unique.

Run the Cypher statement to create the constraint:

**cypher**

Copy to ClipboardRun in Sandbox

CREATE CONSTRAINT Person\_tmdbId

FOR (x:Person)

REQUIRE x.tmdbId IS UNIQUE

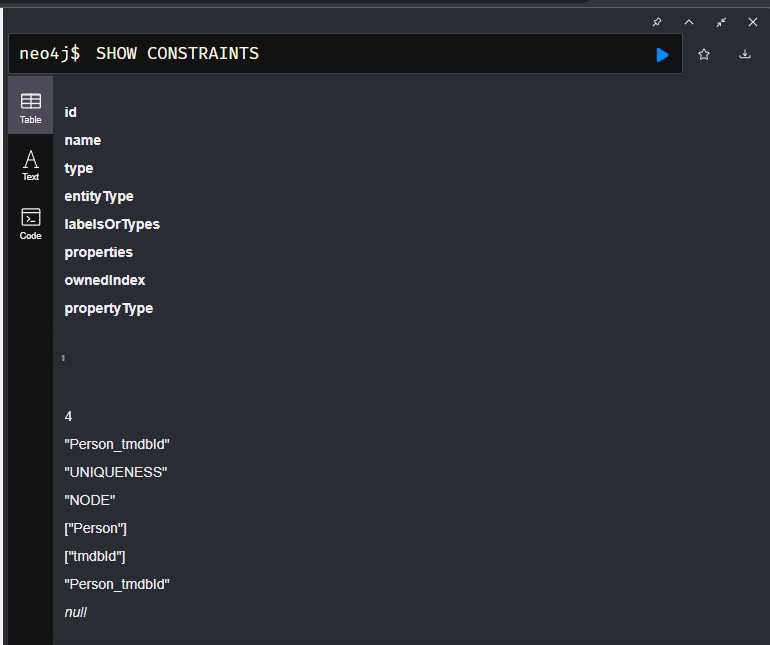


You can check that the constraint has been created by running the following Cypher statement:

**cypher**

Copy to ClipboardRun in Sandbox

SHOW CONSTRAINTS



You should see the constraint named Person\_tmdbId in the results.

If you try to create a Person node with a duplicate tmdbId property value, Neo4j will raise an error.

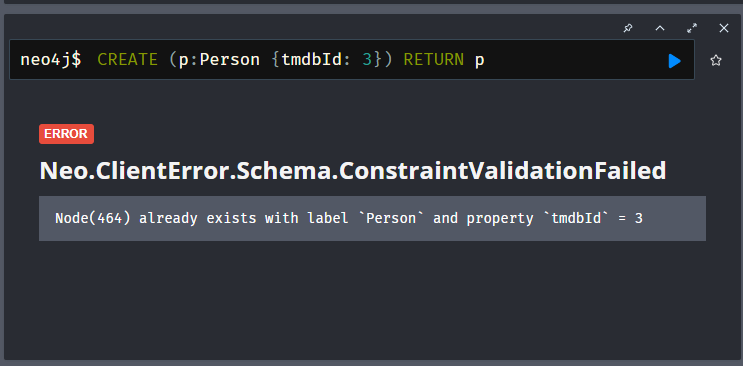
**cypher**

Copy to ClipboardRun in Sandbox

CREATE (p:Person {tmdbId: 3}) RETURN p

Copy to Clipboard

Node(0) already exists with label `Person` and property `tmdbId` = 3



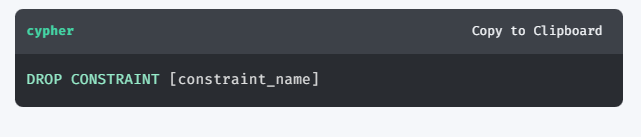
**DROP A CONSTRAINT**

If you need to drop a constraint, use the following Cypher statement.

**cypher**

Copy to Clipboard

DROP CONSTRAINT [constraint\_name]



# Adding Movie Nodes

To complete this challenge, you will need to:

1. Review the [**movies.csv**](https://data.neo4j.com/importing-cypher/movies.csv) file and the data it contains.
2. Create a constraint to ensure that there are no duplicate movies.
3. Construct a Cypher statement to load the CSV file and create the Movie nodes.

## **Movie data**

Open and review the [**movies.csv**](https://data.neo4j.com/importing-cypher/movies.csv) CSV file. The file uses a comma (,) as a field delimiter and it contains headers and data for the following fields:

* movieId
* title
* budget
* countries
* movie\_imdbId
* imdbRating
* imdbVotes
* languages
* plot
* movie\_poster
* released
* revenue
* runtime
* movie\_tmdbId
* movie\_url
* year
* genres

## **Creating the constraint**

The movieId field is unique for each movie.

You should create a constraint named Movie\_movieId to ensure the movieId property is unique for each Movie node.

Here is the Cypher statement you used to create the Person constraint:

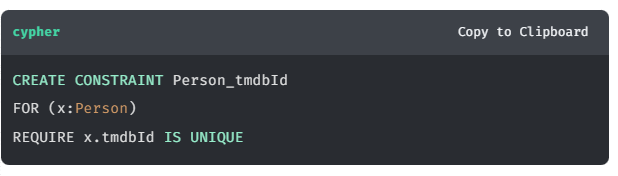
**cypher**

Copy to Clipboard

CREATE CONSTRAINT Person\_tmdbId

FOR (x:Person)

REQUIRE x.tmdbId IS UNIQUE



## **Load the Movie nodes**

You should construct a Cypher to:

1. Load the [**movies.csv**](https://data.neo4j.com/importing-cypher/movies.csv) CSV file from [**https://data.neo4j.com/importing-cypher/movies.csv**](https://data.neo4j.com/importing-cypher/movies.csv)
2. Create the Movie nodes using MERGE with the movieId field as the unique identifier.
3. Set the following properties on the Movie nodes:
   * tmdbId
   * imdbId
   * released
   * title
   * year
   * plot
   * budget

Here is the Cypher statement you used to create the Person nodes:

**cypher**

Copy to Clipboard

LOAD CSV WITH HEADERS FROM 'https://data.neo4j.com/importing-cypher/persons.csv' AS row

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId),

p.bornIn = row.bornIn,

p.name = row.name,

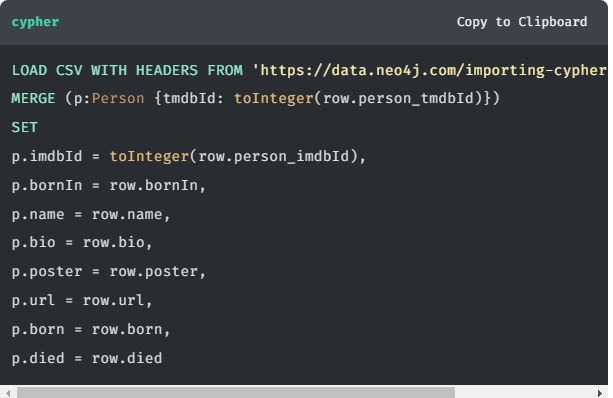
p.bio = row.bio,

p.poster = row.poster,

p.url = row.url,

p.born = row.born,

p.died = row.died



**ADDITIONAL FIELDS AND DATA TYPES**

Later in the course, you modify the import process to include the additional fields and cast relevant data types.

The import should create 93 Movie nodes.

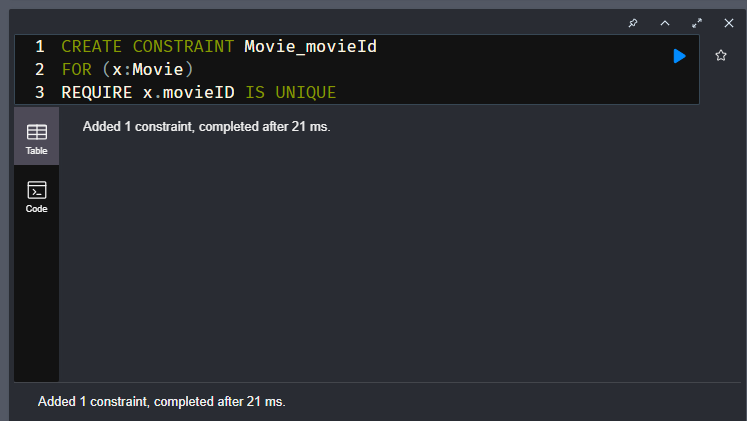
It is a good idea to inspect the data after import. Run the following Cypher statement for return the first 25 Movie nodes:

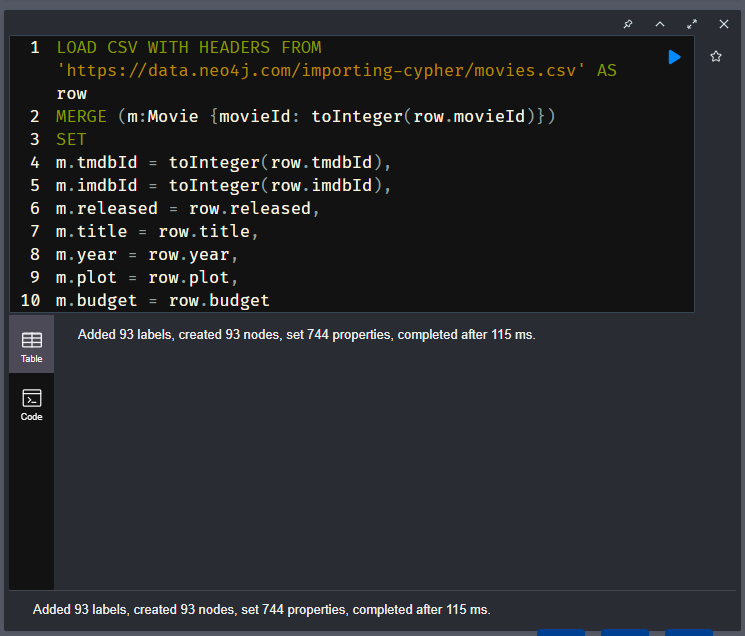
**cypher**

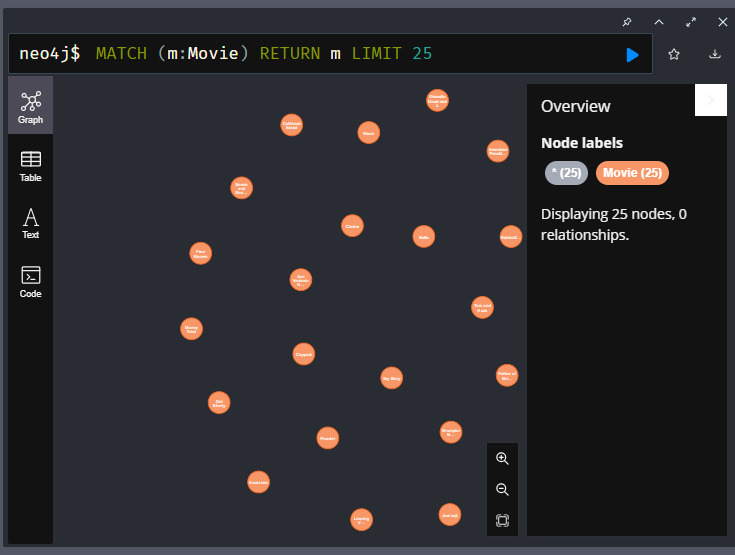
Copy to ClipboardRun in Sandbox

MATCH (m:Movie) RETURN m LIMIT 25

Response-







# Relationships

Youtube video: <https://youtu.be/QqF_-IlWWcA>

Transcript-

In this lesson, you will:

* Learn how to create relationships based on data in a CSV file
* Create ACTED\_IN relationships between the Person and Movie nodes.

## Creating relationships

The [acted\_in.csv](https://data.neo4j.com/importing-cypher/acted_in.csv) file contains the following data:

* movieId - the movieId property of the Movie node
* person\_tmdbId - the tmbdId property of the Person node
* role - the role the person played in the movie

To create the ACTED\_IN relationship, you will need to:

1. LOAD CSV data from the [acted\_in.csv](https://data.neo4j.com/importing-cypher/acted_in.csv) file.
2. MATCH the relevant Person and Movie nodes.
3. Use MERGE to create the ACTED\_IN relationship between the matched Person and Movie nodes.

Review the Cypher statement below, which creates the ACTED\_IN relationships between the Person and Movie nodes.

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

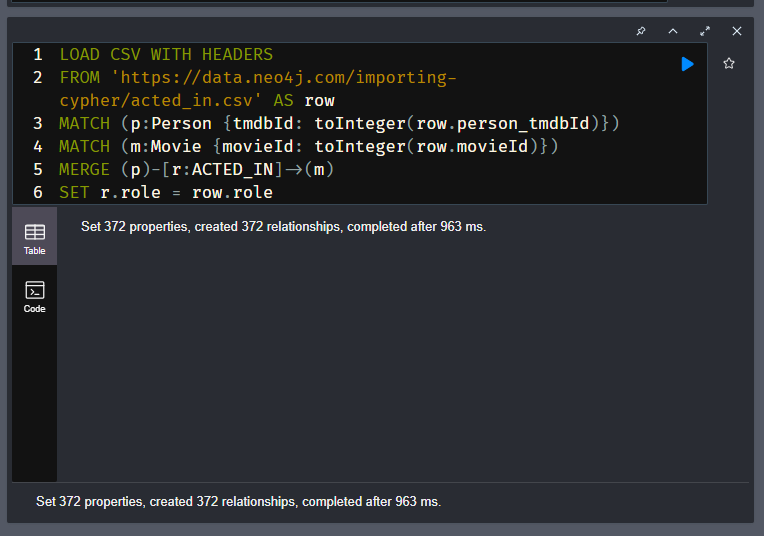
FROM 'https://data.neo4j.com/importing-cypher/acted\_in.csv' AS row

MATCH (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

MATCH (m:Movie {movieId: toInteger(row.movieId)})

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

SET r.role = row.role



Try to identify:

* The 2 MATCH clauses that find the appropriate Person and Movie nodes using the movieId and person\_tmdbId properties.
* The MERGE clause that creates the ACTED\_IN relationship between the matched p and m nodes
* The SET clause that sets the role property of the ACTED\_IN relationship r.

This approach of `MATCH`ing 2 nodes and `MERGE`ing a relationship between them is a typical Cypher pattern.

## ACTED\_IN relationship

Run the Cypher statement to create the ACTED\_IN relationships.

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/acted\_in.csv' AS row

MATCH (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

MATCH (m:Movie {movieId: toInteger(row.movieId)})

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

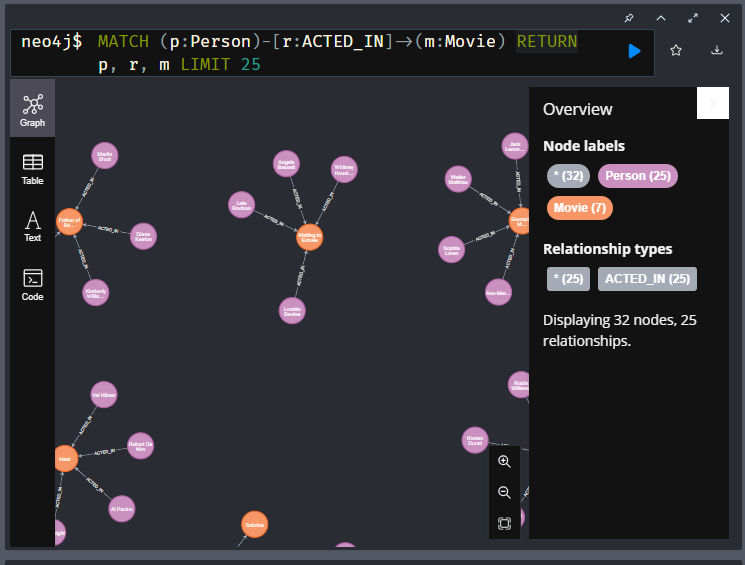
SET r.role = row.role

To verify that you created the ACTED\_IN relationships successfully, run the following Cypher statement to find people who acted in movies:

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie) RETURN p, r, m LIMIT 25



# DIRECTED relationship

To complete this challenge, you will need to create the DIRECTED relationships between the Person and Movie nodes.

## **DIRECTED relationships**

Your challenge is to write a Cypher statement to create the DIRECTED relationship between the Person and Movie nodes.

The [directed.csv](https://data.neo4j.com/importing-cypher/directed.csv) file contains just the movieId and person\_tmdbId IDs.

Unlike the ACTED\_IN relationship, the DIRECTED relationship has no properties (e.g. role), so you will not need to use the SET clause.

**ACTED\_IN RELATIONSHIP**

Here is the Cypher you used to create to the ACTED\_IN relationship.

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/acted\_in.csv' AS row

MATCH (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

MATCH (m:Movie {movieId: toInteger(row.movieId)})

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

SET r.role = row.role

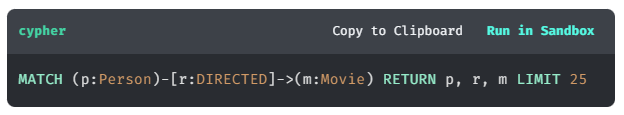


To verify that you created the DIRECTED relationship successfully, run the following Cypher statement:

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (p:Person)-[r:DIRECTED]->(m:Movie) RETURN p, r, m LIMIT 25



**DIRECTED AND ACTED IN**

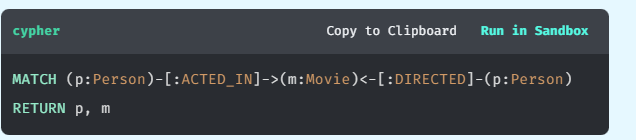
Once you have created the ACTED\_IN and DIRECTED relationships, you can use the data to find people who directed and acted in the same movie.

**cypher**

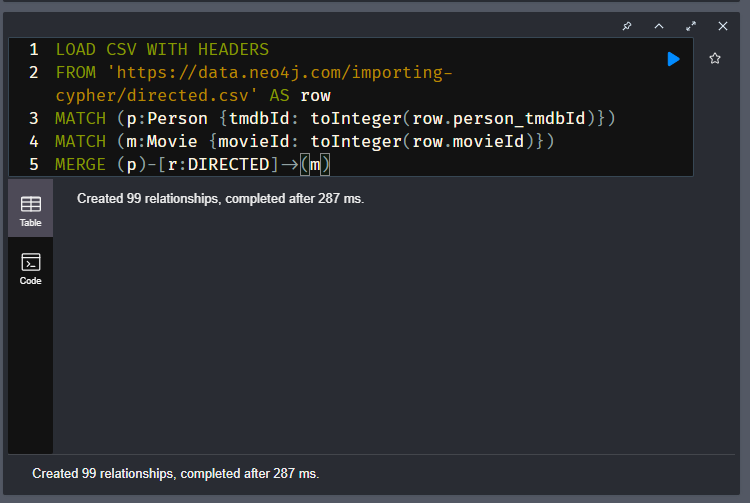
Copy to ClipboardRun in Sandbox

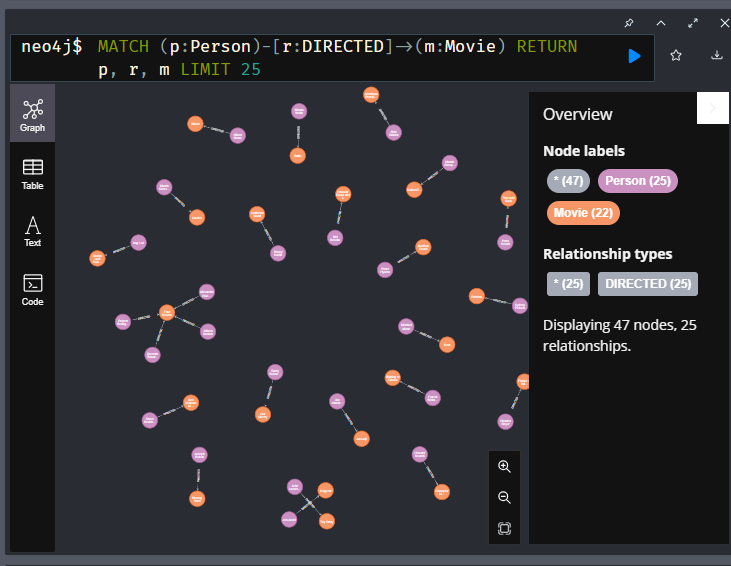
MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(p:Person)

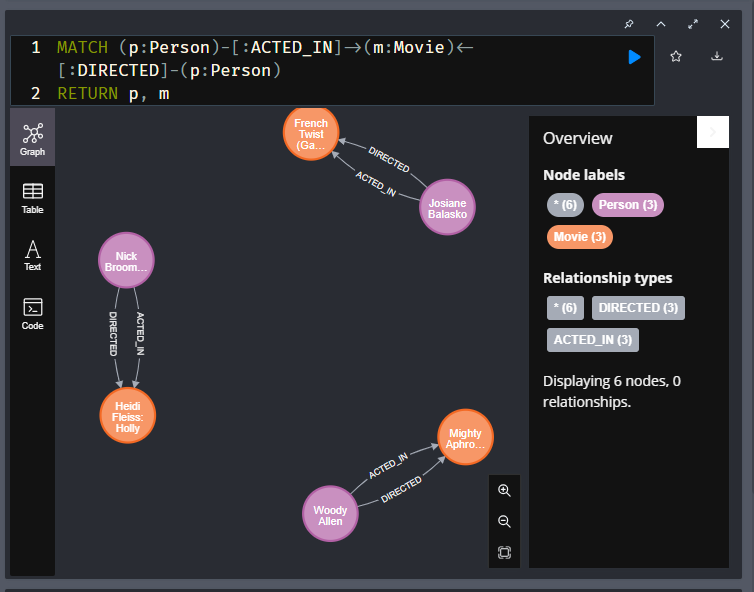
RETURN p, m



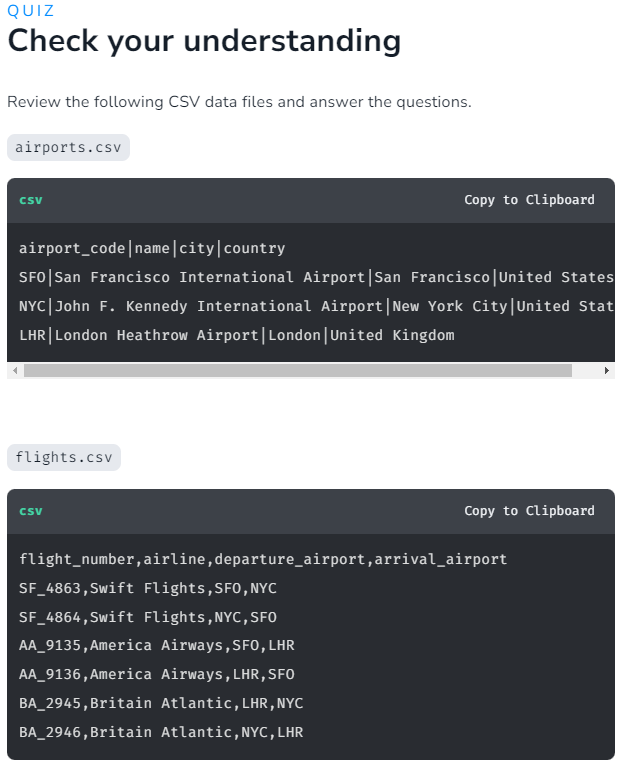
Solution-





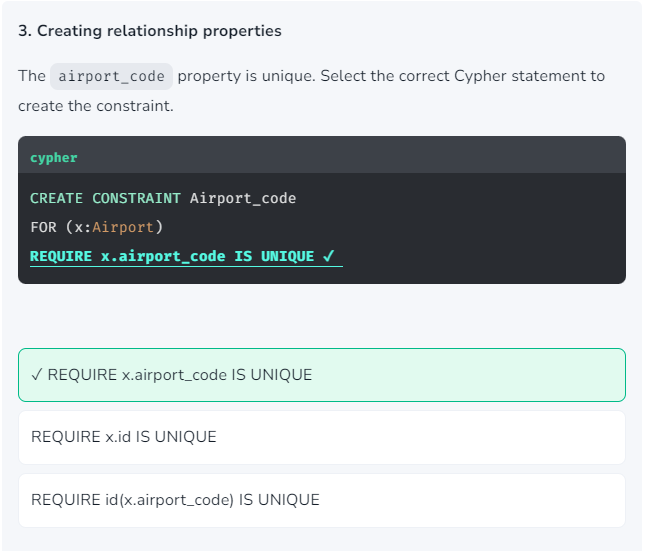


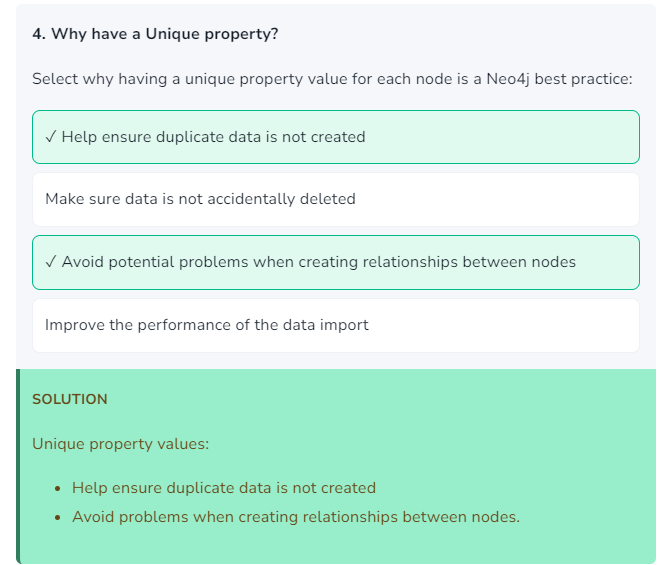
# Quiz: Check your understanding











# Module 3: Data Types, Lists, and Labels

## **Module Overview**

In this module, you will learn about:

* Neo4j data types and how to cast CSV data to the correct type
* List properties and how to create them from CSV data
* Adding multiple labels to nodes

## References

* [**Neo4j Sandbox**](https://sandbox.neo4j.com/)
* [**MERGE**](https://neo4j.com/docs/cypher-manual/current/clauses/merge/)
* [**LOAD CSV**](https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/)
* [**SET**](https://neo4j.com/docs/cypher-manual/current/clauses/set/)
* [**split()**](https://neo4j.com/docs/cypher-manual/current/functions/string/#functions-split)
* [**Values and Types**](https://neo4j.com/docs/cypher-manual/current/values-and-types/)

# Data Types

Youtube video: <https://youtu.be/zgCYIKR_g3Q>

Transcript-

In this lesson, you will explore how to cast data from a CSV file to different data types in Neo4j.

## **Casting**

All data loaded using LOAD CSV will be returned as strings - you need to cast the data to an appropriate data type before being written to a property.

The types of data that you can store as properties in Neo4j include:

* String
* Integer
* Float (decimal values)
* Boolean
* Date/Datetime
* Point (spatial)
* Lists of values

There are Cypher functions to cast data to appropriate types. For example, when creating the Person nodes, you used the toInteger() function to cast IDs to integers.

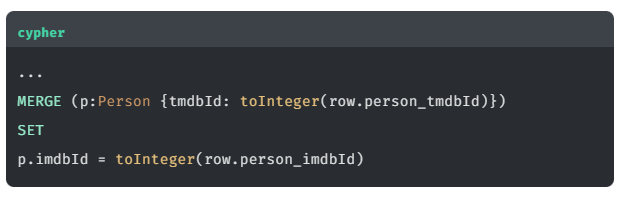
**cypher**

...

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId)



Cypher functions to cast data include:

| **Function** | **Description** |
| --- | --- |
| toBoolean() | Converts a string to a boolean value |
| toFloat() | Converts a string to a float value |
| toInteger() | Converts a string to an integer value |
| toString() | Converts a value to a string |
| date() | Converts a string to a date value |
| datetime() | Converts a string to a date and time value |

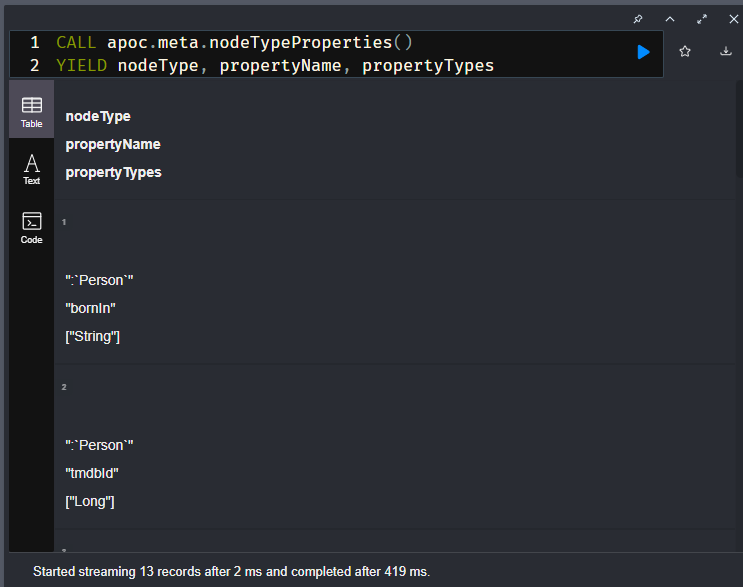
You can use the apoc.meta.nodeTypeProperties() function to show the data types used in the graph:

**cypher**

Copy to ClipboardRun in Sandbox

CALL apoc.meta.nodeTypeProperties()

YIELD nodeType, propertyName, propertyTypes



Review the results and note, except for the IDs, that the data types for properties of Person are all strings.

| **Node Type** | **Property** | **Data Type** |
| --- | --- | --- |
| ":`Person`" | "tmdbId" | ["Long"] |
| ":`Person`" | "imdbId" | ["Long"] |
| ":`Person`" | "bornIn" | ["String"] |
| ":`Person`" | "born" | ["String"] |
| ":`Person`" | "name" | ["String"] |
| ":`Person`" | "bio" | ["String"] |
| ":`Person`" | "died" | ["String"] |

Neo4j will return the data type Long for integer values.

### Person node dates

The Person nodes born and died properties are both dates, not strings.

You used this Cypher statement to create the Person nodes:

**cypher**

Copy to Clipboard

LOAD CSV WITH HEADERS FROM 'https://data.neo4j.com/importing-cypher/persons.csv' AS row

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId),

p.bornIn = row.bornIn,

p.name = row.name,

p.bio = row.bio,

p.poster = row.poster,

p.url = row.url,

p.born = row.born,

p.died = row.died

It should be modified to use the date() function to convert the born and died properties to Date values.

## Correct the Person nodes

Run this updated query to modify the born and died properties to be Date values.

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS FROM 'https://data.neo4j.com/importing-cypher/persons.csv' AS row

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId),

p.bornIn = row.bornIn,

p.name = row.name,

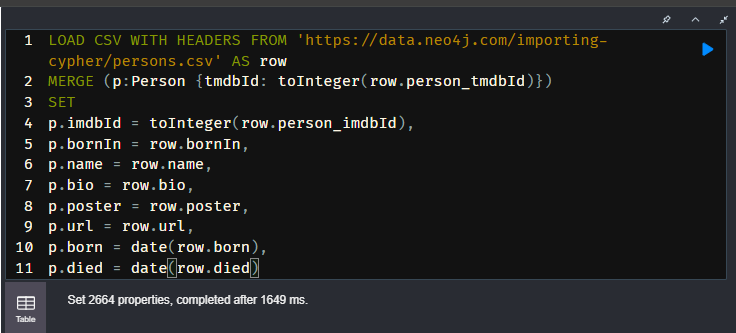
p.bio = row.bio,

p.poster = row.poster,

p.url = row.url,

p.born = date(row.born),

p.died = date(row.died)



**USING MERGE NOT CREATE?**

As MERGE was used in this Cypher statement, you can run it multiple times without creating duplicate nodes. It will update the existing nodes with the new date values. If you used CREATE instead, you would create new nodes each time you ran the statement.

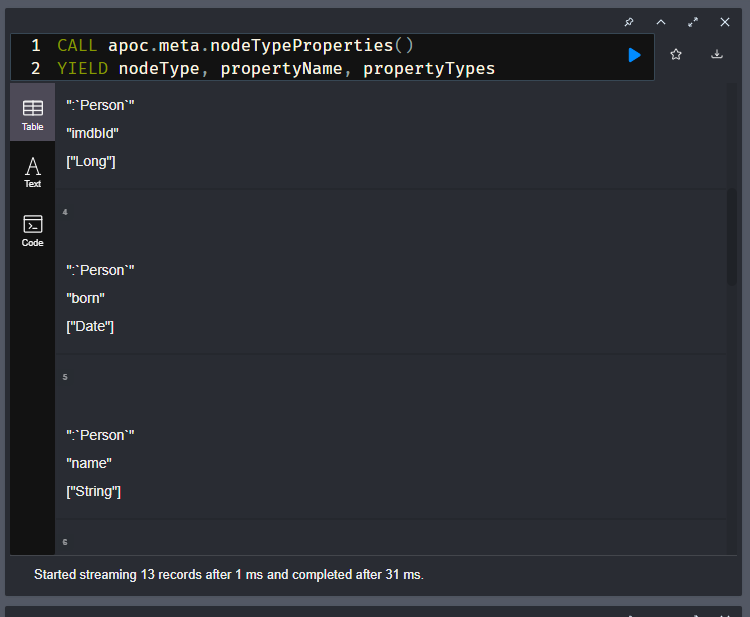
Use the apoc.meta.nodeTypeProperties function again to check that the born and died properties are now Date values:

**cypher**

Copy to ClipboardRun in Sandbox

CALL apoc.meta.nodeTypeProperties()

YIELD nodeType, propertyName, propertyTypes



**ADVANTAGES OF USING**DATE

The Date data type allows you to extract the year, month, and day from the date. For example,

**Cypher**

Copy to ClipboardRun in Sandbox

MATCH (p:Person)

RETURN p.born.year as YearOfBirth



The remaining properties are all string values, so casting them to a different data type is unnecessary.

# Adding Properties

## Challenge-

In this challenge, you will update the import to add new properties to the Movie nodes and cast them to the correct type.

Previously, you used the following Cypher statement to import the Movie nodes:

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/movies.csv'

AS row

MERGE (m:Movie {movieId: toInteger(row.movieId)})

SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

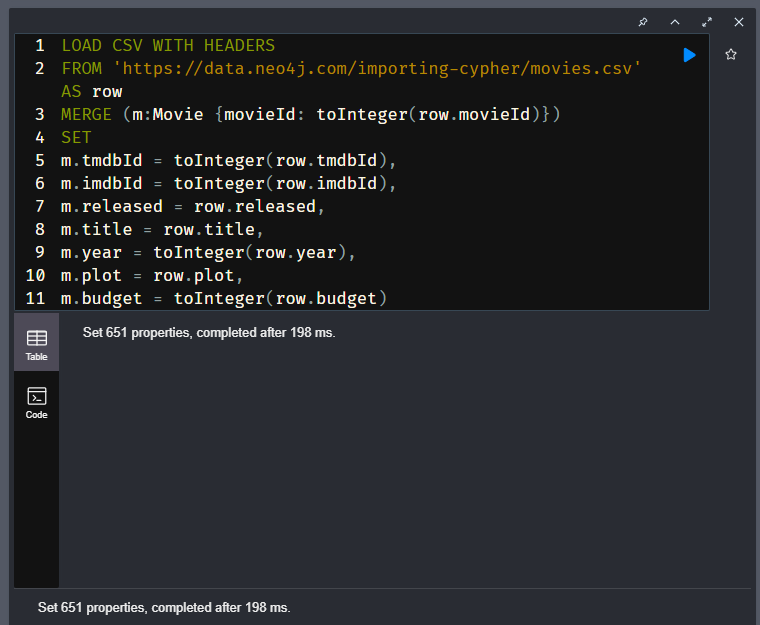
m.released = row.released,

m.title = row.title,

m.year = row.year,

m.plot = row.plot,

m.budget = row.budget



The year and budget properties are not cast and therefore stored as strings.

Update and run the existing import to cast the year and budget properties to integers using the toInteger() function.

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/movies.csv'

AS row

MERGE (m:Movie {movieId: toInteger(row.movieId)})

SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

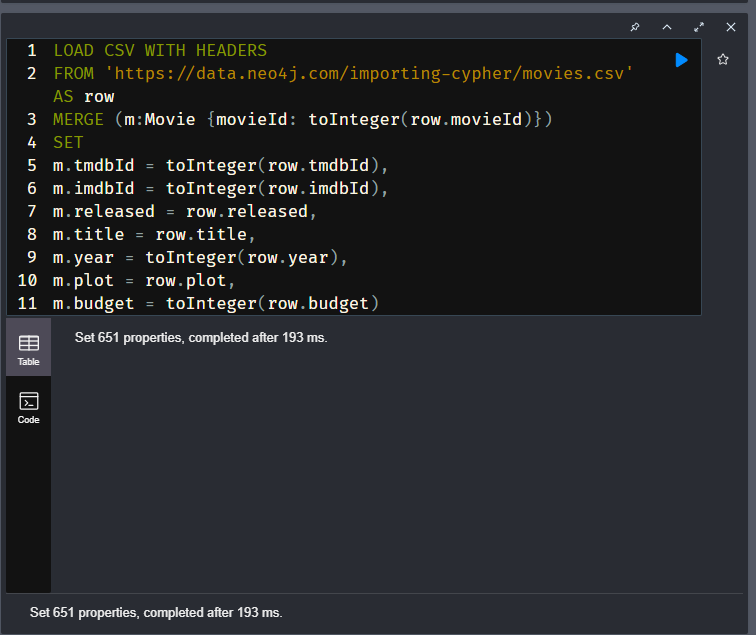
m.released = row.released,

m.title = row.title,

m.year = toInteger(row.year),

m.plot = row.plot,

m.budget = toInteger(row.budget)



The [movies.csv](https://data.neo4j.com/importing-cypher/movies.csv) file also contains the following additional properties:

* imdbRating
* poster
* runtime
* imdbVotes
* revenue
* url

Review the data in the file and identify the data type and cast function for each property.

Reveal the data types and functions

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Finally, update the import to add these properties to the Movie nodes and cast them to the correct type.

## Solution-

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/movies.csv' AS row

MERGE (m:Movie {movieId: toInteger(row.movieId)})

SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

m.released = row.released,

m.title = row.title,

m.year = toInteger(row.year),

m.plot = row.plot,

m.budget = toInteger(row.budget),

m.imdbRating = toFloat(row.imbdRating),

m.poster = row.poster,

m.runtime = toInteger(row.runtime),

m.imdbVotes =toInteger(row.imdbVotes),

m.revenue = toInteger(row.revenue),

m.url = row.url



# Lists

In this lesson, you will learn about:

* Multi-value properties and lists
* How to transform a string value into a list
* The split function

## **What is a Multi-value property?**

A multi-value property is a property that can hold one or more values. Neo4j represents this type of data as a list (or "StringArray").

All values in a list must have the same data type. For example:

* ["Apple", "Banana, "Orange"]
* [100, 55, 4]

The [movies.csv](https://data.neo4j.com/importing-cypher/movies.csv) data file contains multi-value properties, including:

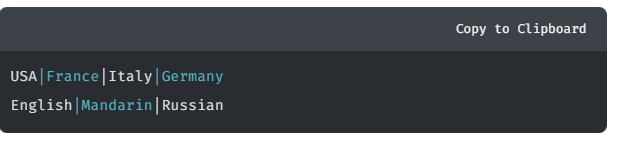
* countries - the countries which produced the movie
* languages - the languages spoken in the movie

Find the countries and languages data in the movies.csv file. You will see that each property contains a list of values separated by a | character. For example:

Copy to Clipboard

USA|France|Italy|Germany

English|Mandarin|Russian



## **Split the data into a list**

The split function will transform a string value into a list. The split function takes two arguments:

* The string to split
* The character to split on

This updated Movie import creates a list of countries by using the split function:

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/movies.csv'

AS row

MERGE (m:Movie {movieId: toInteger(row.movieId)})

SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

m.released = date(row.released),

m.title = row.title,

m.year = toInteger(row.year),

m.plot = row.plot,

m.budget = toInteger(row.budget),

m.imdbRating = toFloat(row.imdbRating),

m.poster = row.poster,

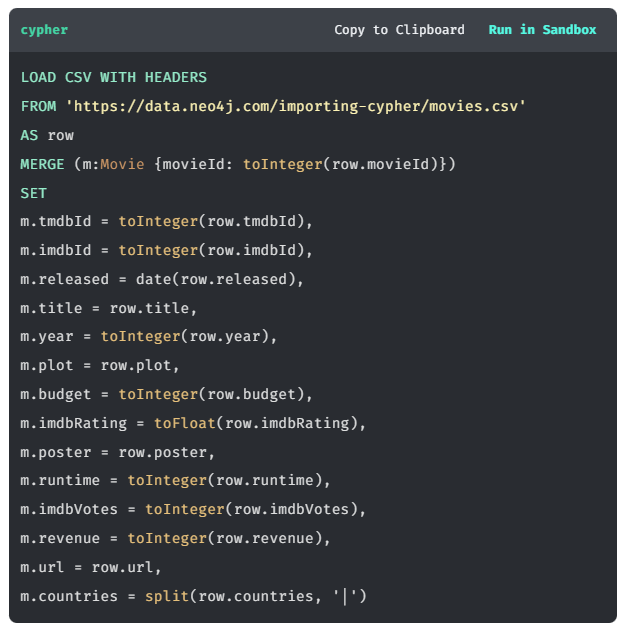
m.runtime = toInteger(row.runtime),

m.imdbVotes = toInteger(row.imdbVotes),

m.revenue = toInteger(row.revenue),

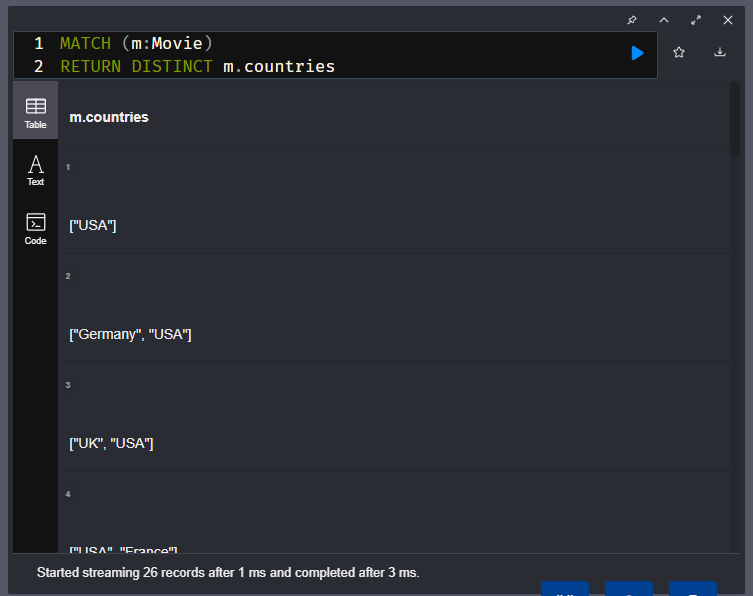
m.url = row.url,

m.countries = split(row.countries, '|')



The statement sets the countries property as a list by splitting the data from the CSV file by the | character.

Run the query and inspect the countries property data by writing a MATCH query to return the data.



You can query data in a list using the IN operator. For example, finding all the movies where "French" is a listed language.

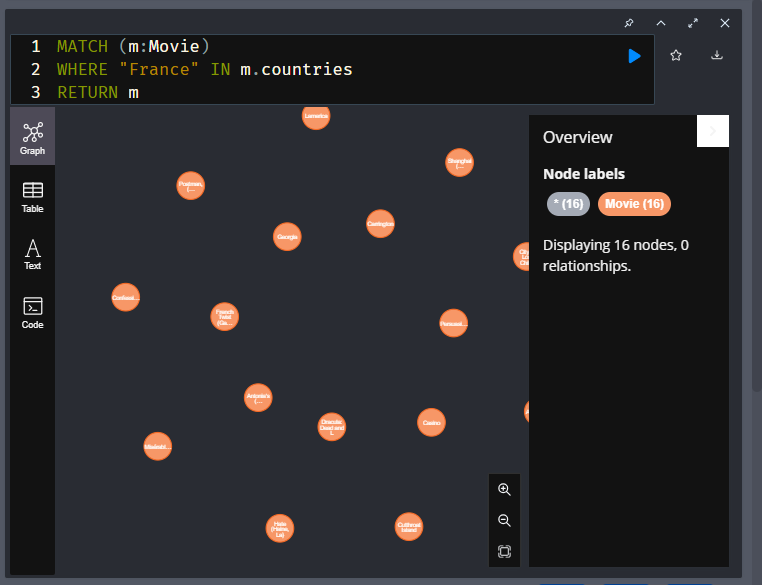
**cypher**

Copy to ClipboardRun in Sandbox

MATCH (m:Movie)

WHERE "France" IN m.countries

RETURN m



# Add the languages property

## Challenge-

Your challenge is to modify this Cypher query to add the languages property.

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/movies.csv'

AS row

MERGE (m:Movie {movieId: toInteger(row.movieId)})

SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

m.released = date(row.released),

m.title = row.title,

m.year = toInteger(row.year),

m.plot = row.plot,

m.budget = toInteger(row.budget),

m.imdbRating = toFloat(row.imdbRating),

m.poster = row.poster,

m.runtime = toInteger(row.runtime),

m.imdbVotes = toInteger(row.imdbVotes),

m.revenue = toInteger(row.revenue),

m.url = row.url,

m.countries = split(row.countries, '|')

The languages property should be the list of languages in the CSV file. As with countries each language is separated by the (|) character.

## Solution-

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/movies.csv' AS row

MERGE (m:Movie {movieId: toInteger(row.movieId)})

SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

m.released = date(row.released),

m.title = row.title,

m.year = toInteger(row.year),

m.plot = row.plot,

m.budget = toInteger(row.budget),

m.imdbRating = toFloat(row.imdbRating),

m.poster = row.poster,

m.runtime = toInteger(row.runtime),

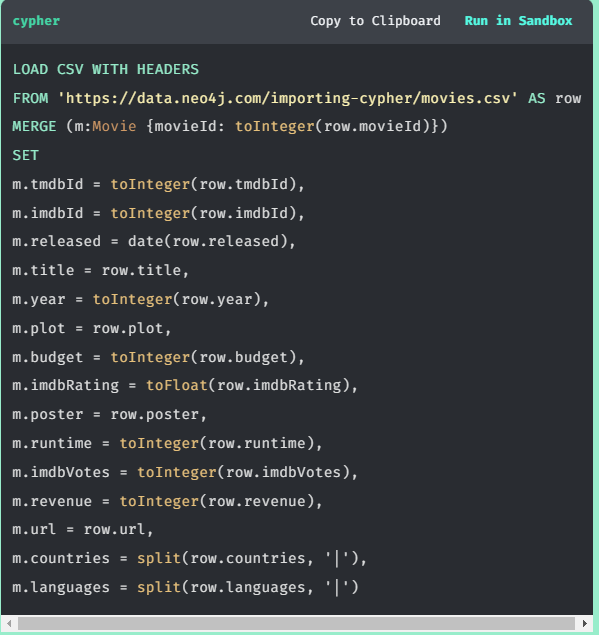
m.imdbVotes = toInteger(row.imdbVotes),

m.revenue = toInteger(row.revenue),

m.url = row.url,

m.countries = split(row.countries, '|'),

m.languages = split(row.languages, '|')



# Labels

Youtube video: <https://youtu.be/6AfqYU55img>

Transcript-

In this lesson, you will learn how to add additional labels to existing nodes.

## **Add Labels to Existing Nodes**

Adding labels to existing nodes can make your graph more useful and performant.

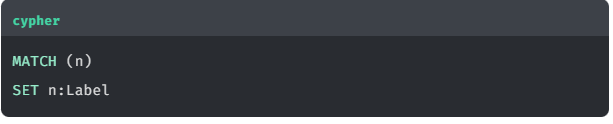
The Person nodes in the graph represent both actors and directors. To determine if a person is an actor or director, you need to query the ACTED\_IN or DIRECTED relationships. Alternatively, you could add labels to the existing nodes to distinguish between actors and directors.

You can add labels to a node using SET, the syntax is:

**cypher**

MATCH (n)

SET n:Label



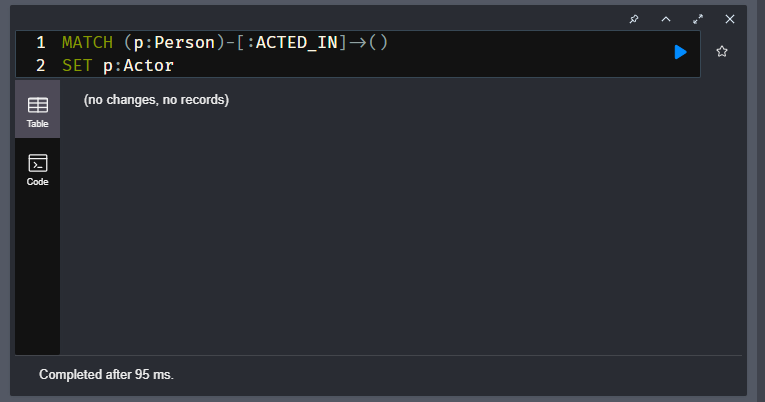
The following query would add the Actor label to all Person nodes that have acted in a movie:

**cypher**

Copy to ClipboardRun in Sandbox

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

SET p:Actor



The query:

1. Finds all the Person nodes with an ACTED\_IN relationship to a node
2. Sets the Actor label to those nodes

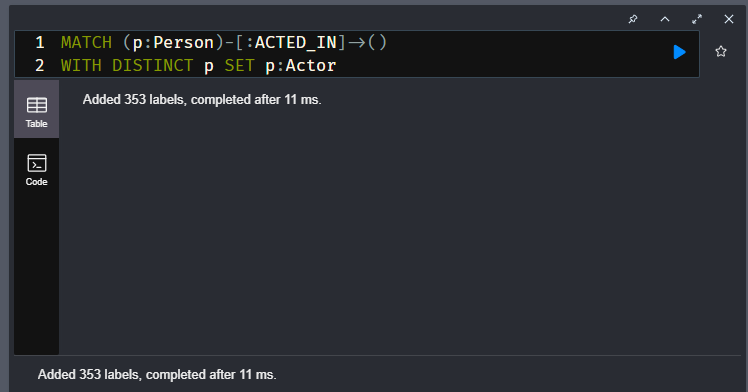
As there are people in the database who have acted in more than one movie, you can use WITH DISTINCT to ensure that each person is only labeled once. Although not essential, this will improve the performance of the statement.

**cypher**

Copy to ClipboardRun in Sandbox

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

WITH DISTINCT p SET p:Actor



## **Add Actor label**

Run the query to add the Actor label:

**cypher**

Copy to ClipboardRun in Sandbox

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

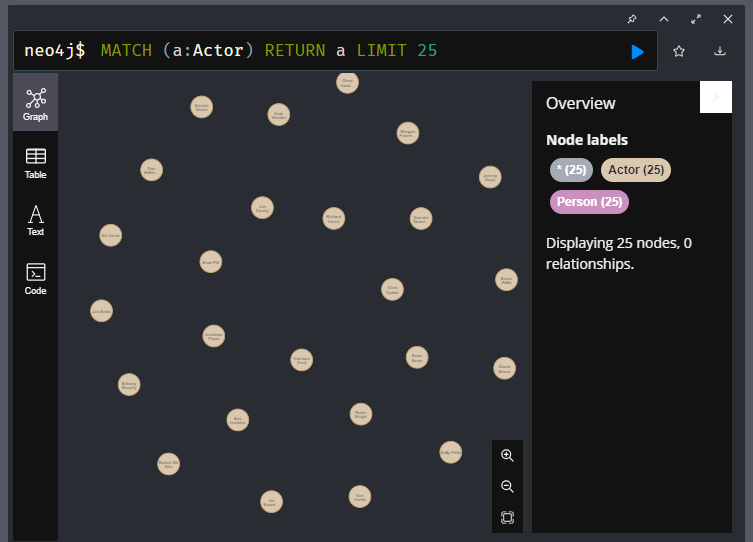
WITH DISTINCT p SET p:Actor

You can confirm it was successful by using the Actor label to find actors in the graph:

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (a:Actor) RETURN a LIMIT 25



**FILTERING BY LABELS IS FASTER**

By adding the Actor label to the graph, queries that use the label rather than the relationship will be quicker to return.

# Add Director label

## Challenge-

Your challenge is to create a Cypher statement that adds the Director label to all Person nodes with a DIRECTED relationship to a Movie node.

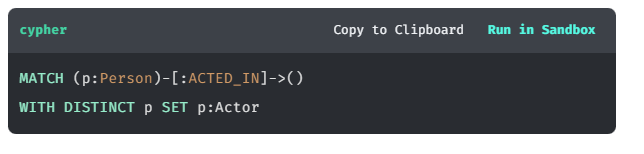
Previously, you used the following Cypher to add the Actor label to nodes with the ACTED\_IN relationship.

**cypher**

Copy to ClipboardRun in Sandbox

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

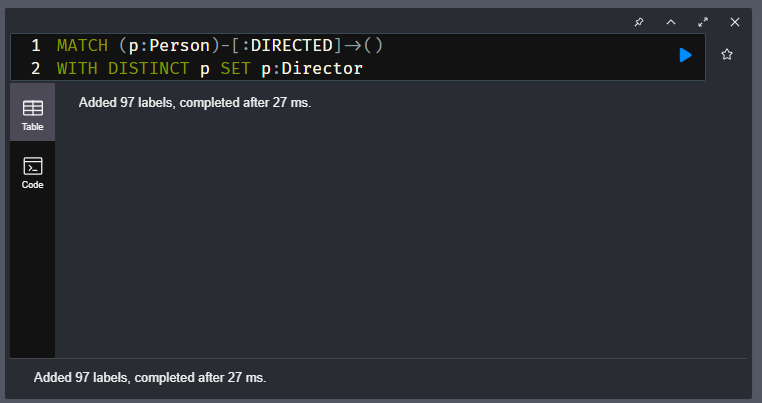
WITH DISTINCT p SET p:Actor



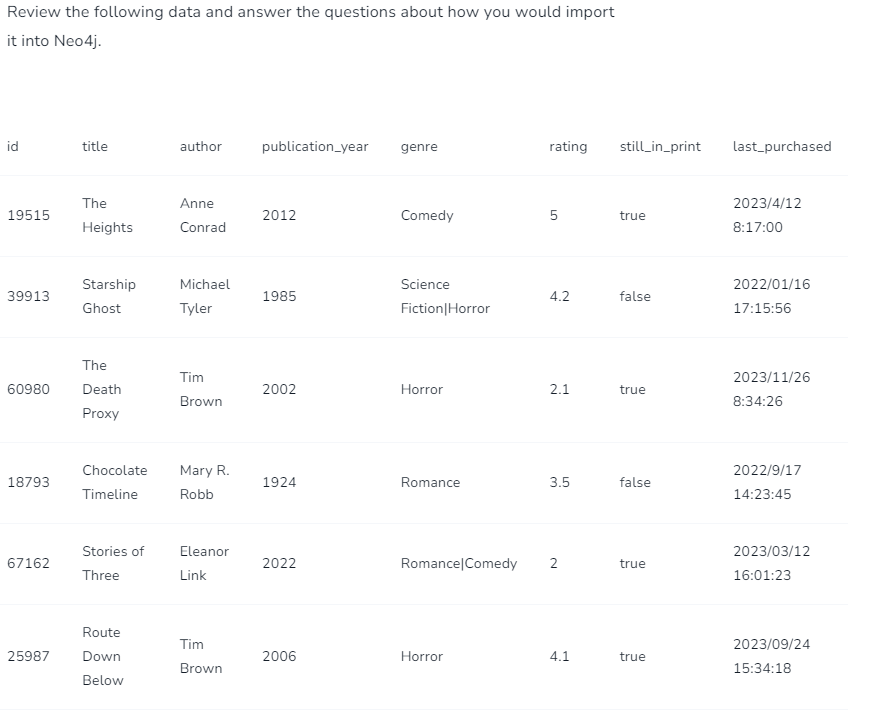
## Solution-

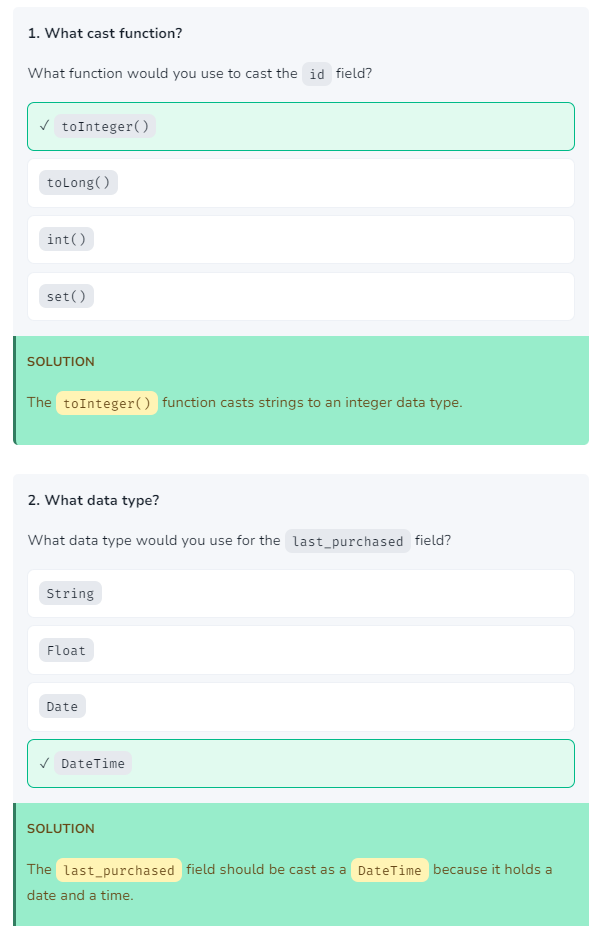
MATCH (p:Person)-[:DIRECTED]->()

WITH DISTINCT p SET p:Director;

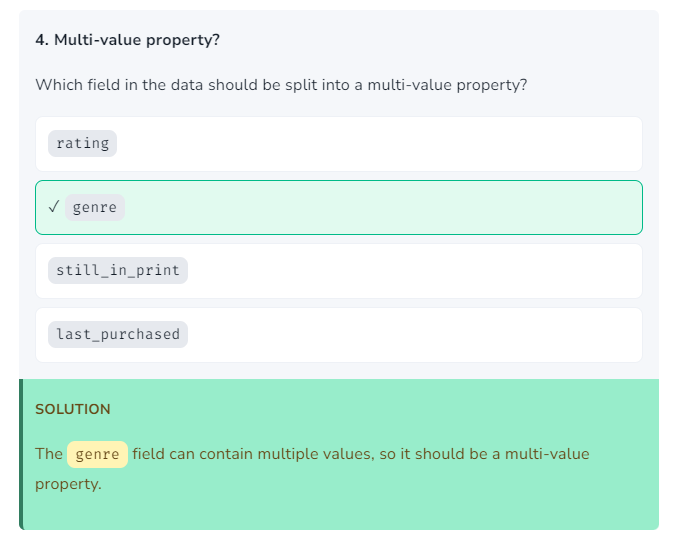


# Quiz: Check your understanding









# Module 4: Importing data considerations

## Module Overview

In this module, you will learn about:

* The considerations you need to make when importing data into Neo4j
* Strategies for importing data into Neo4j, including making multiple passes over the data
* Why and how to split an import into transactions
* The next steps you can take to continue your learning

## References

* [**Neo4j Sandbox**](https://sandbox.neo4j.com/)
* [neo4j-admin import**command line interface**](https://neo4j.com/docs/operations-manual/current/tools/neo4j-admin/neo4j-admin-import/)
* [**Pentaho Data Integration (PDI) / Kettle**](https://www.hitachivantara.com/en-us/products/pentaho-platform/data-integration-analytics/pentaho-tutorials.html)
* [**Eager operator in the Neo4j documentation**](https://neo4j.com/docs/cypher-manual/current/execution-plans/operators/#query-plan-eager)
* [**Avoiding the Eager Operator**](https://www.markhneedham.com/blog/2014/10/23/neo4j-cypher-avoiding-the-eager/)

# Building an Import Process

During this course, you have created queries which complete the following tasks:

* Create Person and Movie constraints
* Import data from persons.csv and create Person nodes
* Import data from movies.csv and create Movie nodes
* Create ACTED\_IN and DIRECTED relationships between Person and Movie nodes
* Create additional ACTOR and DIRECTOR labels on Person nodes

All the queries are independent of each other and do not form a single process.

View all the import queries

**cypher**

Copy to Clipboard

CREATE CONSTRAINT Person\_tmdbId IF NOT EXISTS

FOR (x:Person)

REQUIRE x.tmdbId IS UNIQUE;

CREATE CONSTRAINT Movie\_movieId IF NOT EXISTS

FOR (x:Movie)

REQUIRE x.movieId IS UNIQUE;

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/persons.csv' AS row

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId),

p.bornIn = row.bornIn,

p.name = row.name,

p.bio = row.bio,

p.poster = row.poster,

p.url = row.url,

p.born = date(row.born),

p.died = date(row.died);

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/movies.csv' AS row

MERGE (m:Movie {movieId: toInteger(row.movieId)})

SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

m.released = date(row.released),

m.title = row.title,

m.year = toInteger(row.year),

m.plot = row.plot,

m.budget = toInteger(row.budget),

m.imdbRating = toFloat(row.imdbRating),

m.poster = row.poster,

m.runtime = toInteger(row.runtime),

m.imdbVotes = toInteger(row.imdbVotes),

m.revenue = toInteger(row.revenue),

m.url = row.url,

m.countries = split(row.countries, '|'),

m.languages = split(row.languages, '|');

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/acted\_in.csv' AS row

MATCH (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

MATCH (m:Movie {movieId: toInteger(row.movieId)})

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

SET r.role = row.role;

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/directed.csv' AS row

MATCH (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

MATCH (m:Movie {movieId: toInteger(row.movieId)})

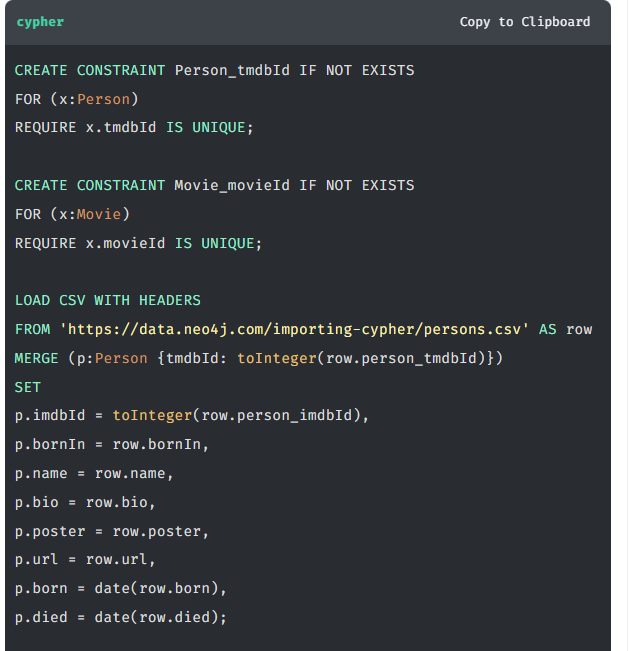
MERGE (p)-[r:DIRECTED]->(m);

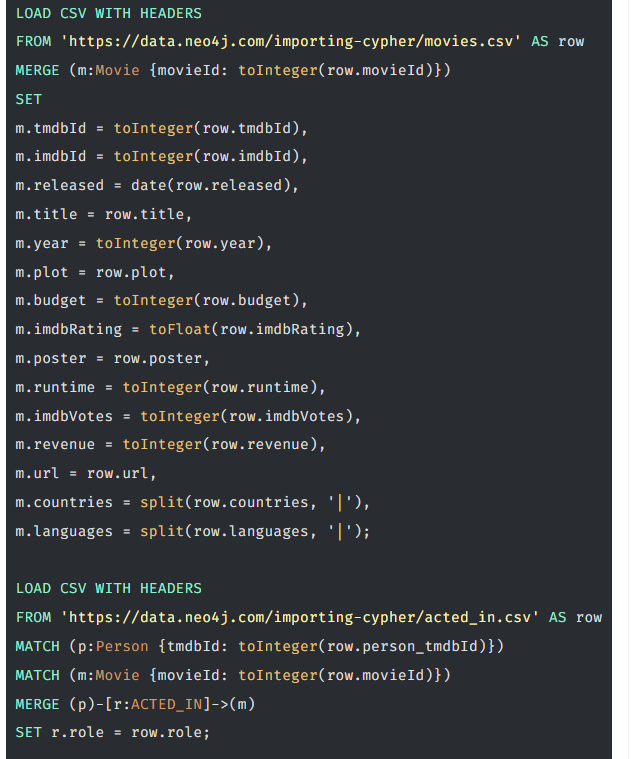
MATCH (p:Person)-[:ACTED\_IN]->()

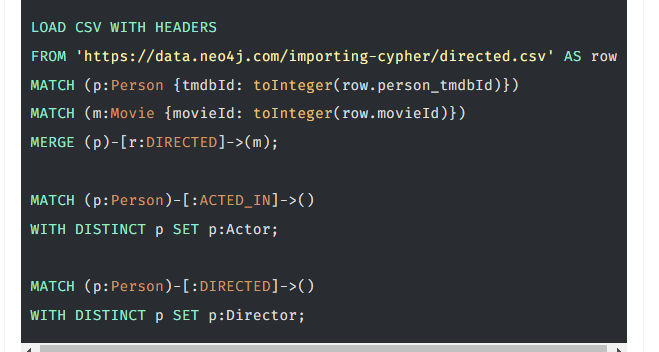
WITH DISTINCT p SET p:Actor;

MATCH (p:Person)-[:DIRECTED]->()

WITH DISTINCT p SET p:Director;







In this lesson, you will build a single import process to rebuild the graph by:

* Deleting any existing data
* Dropping any existing constraints
* Run the queries to create the constraints, nodes, and relationships

The benefit of this approach is that you can easily re-run and test a single repeatable process.

## **Multiple Queries**

To run multiple queries together, you must put a semi-colon (;) at the end of each query.

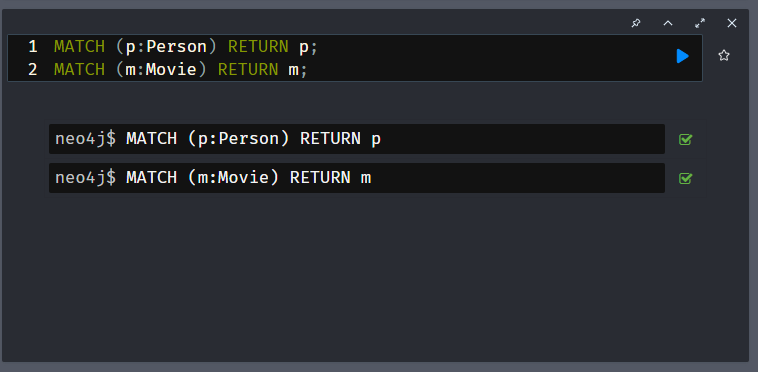
For example, this Cypher code contains two separate queries which will run one after the other:

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (p:Person) RETURN p;

MATCH (m:Movie) RETURN m;



## Resetting the data

Before you can re-run the import process, you must delete any existing data and drop any constraints.

The nodes and relationships within the graph hold all of the data. You will need to delete the relationships before deleting the nodes.

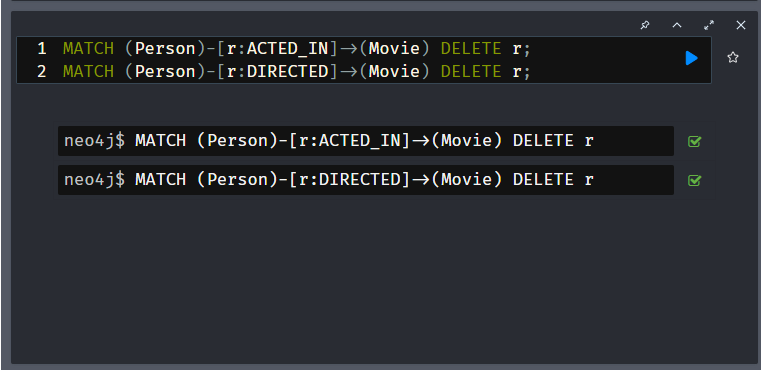
The following Cypher will delete the ACTED\_IN and DIRECTED relationships:

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (Person)-[r:ACTED\_IN]->(Movie) DELETE r;

MATCH (Person)-[r:DIRECTED]->(Movie) DELETE r;



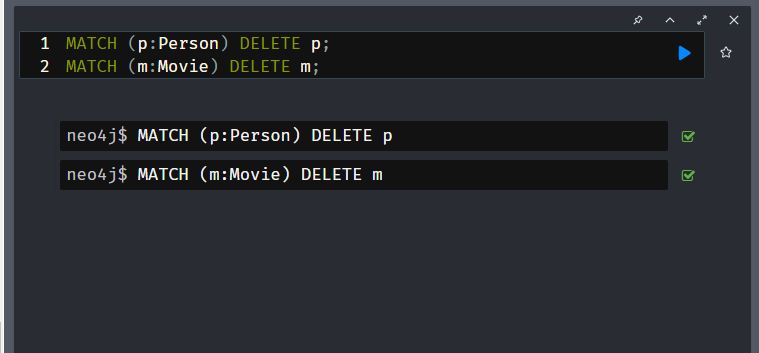
Once the relationships are deleted, you can delete the Person and Movie nodes:

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (p:Person) DELETE p;

MATCH (m:Movie) DELETE m;



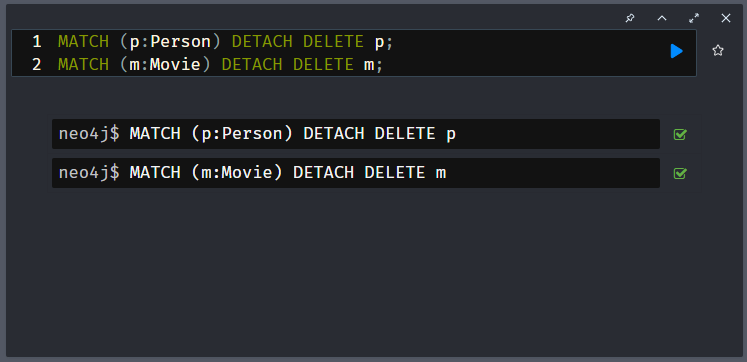
Alternatively, you can use DETACH DELETE to delete the nodes and relationships at the same time:

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (p:Person) DETACH DELETE p;

MATCH (m:Movie) DETACH DELETE m;



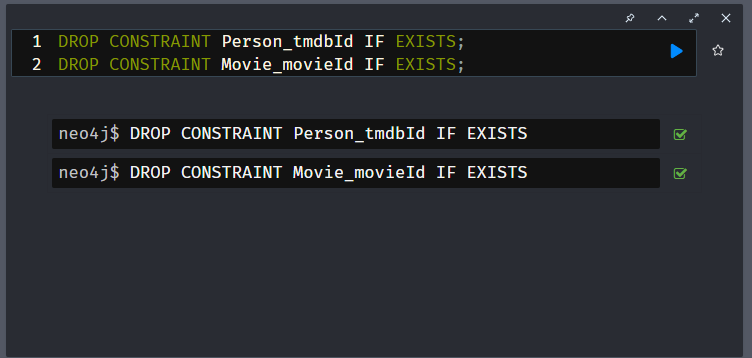
You could also drop the constraints on the Person and Movie nodes if they exist:

**cypher**

Copy to ClipboardRun in Sandbox

DROP CONSTRAINT Person\_tmdbId IF EXISTS;

DROP CONSTRAINT Movie\_movieId IF EXISTS;



These queries reset the database and allow you to re-run the import process.

## **Importing the data**

Combining the queries above with those from the previous lessons will create a single import process.

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (p:Person) DETACH DELETE p;

MATCH (m:Movie) DETACH DELETE m;

DROP CONSTRAINT Person\_tmdbId IF EXISTS;

DROP CONSTRAINT Movie\_movieId IF EXISTS;

CREATE CONSTRAINT Person\_tmdbId IF NOT EXISTS

FOR (x:Person)

REQUIRE x.tmdbId IS UNIQUE;

CREATE CONSTRAINT Movie\_movieId IF NOT EXISTS

FOR (x:Movie)

REQUIRE x.movieId IS UNIQUE;

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/persons.csv' AS row

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId),

p.bornIn = row.bornIn,

p.name = row.name,

p.bio = row.bio,

p.poster = row.poster,

p.url = row.url,

p.born = date(row.born),

p.died = date(row.died);

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/movies.csv' AS row

MERGE (m:Movie {movieId: toInteger(row.movieId)})

SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

m.released = date(row.released),

m.title = row.title,

m.year = toInteger(row.year),

m.plot = row.plot,

m.budget = toInteger(row.budget),

m.imdbRating = toFloat(row.imdbRating),

m.poster = row.poster,

m.runtime = toInteger(row.runtime),

m.imdbVotes = toInteger(row.imdbVotes),

m.revenue = toInteger(row.revenue),

m.url = row.url,

m.countries = split(row.countries, '|'),

m.languages = split(row.languages, '|');

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/acted\_in.csv' AS row

MATCH (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

MATCH (m:Movie {movieId: toInteger(row.movieId)})

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

SET r.role = row.role;

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/directed.csv' AS row

MATCH (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

MATCH (m:Movie {movieId: toInteger(row.movieId)})

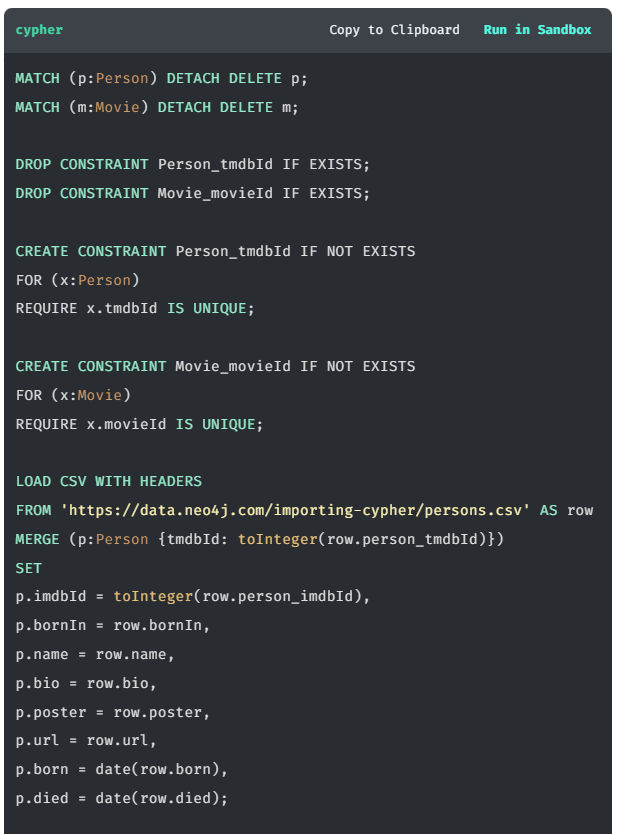
MERGE (p)-[r:DIRECTED]->(m);

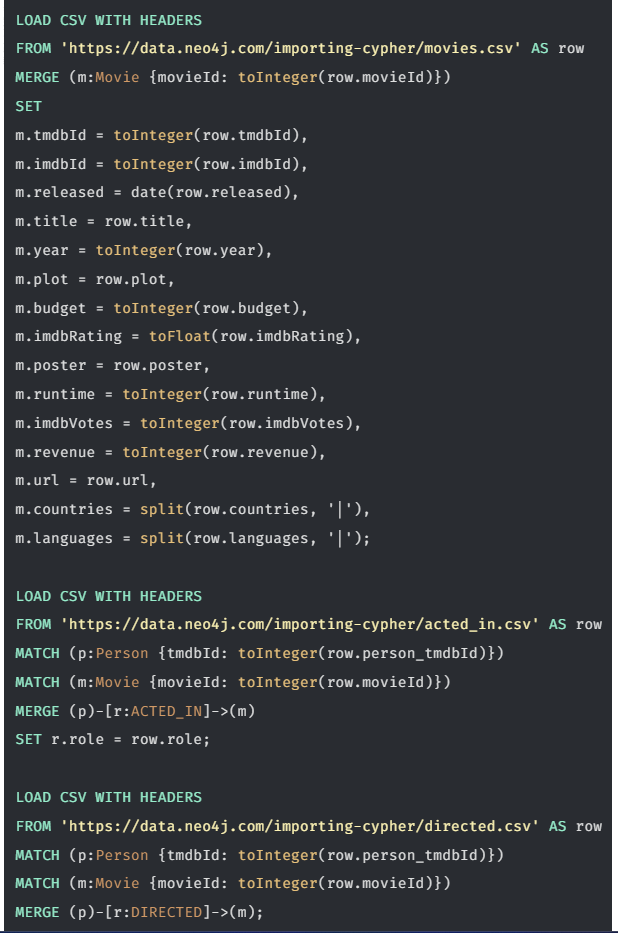
MATCH (p:Person)-[:ACTED\_IN]->()

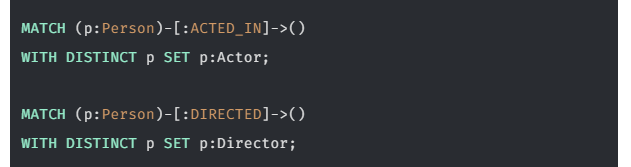
WITH DISTINCT p SET p:Actor;

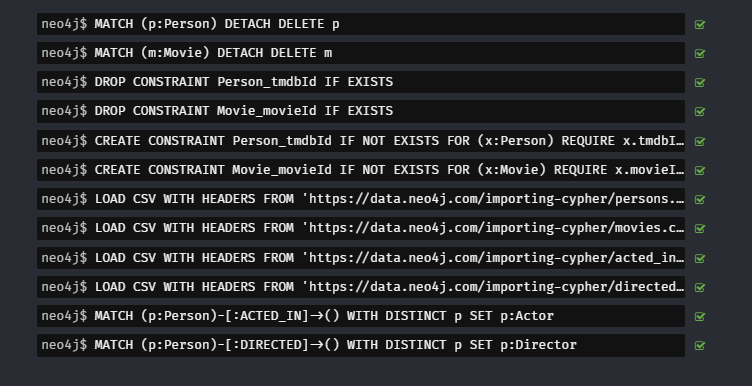
MATCH (p:Person)-[:DIRECTED]->()

WITH DISTINCT p SET p:Director;









You can run this query at any point to refresh the database with the latest data.

A single process to build your graph provides a consistent mechanism to test your import.

# Transactions

The Cypher queries you have written will run within a single transaction. As a result, the data is rolled back if a failure occurs; and the graph is unchanged.

Importing significant volumes of data in a single transaction can result in large write operations - this can cause performance issues and potential failure.

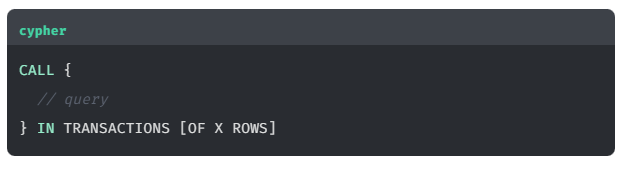
You can split a query into multiple transactions using the CALL clause with IN TRANSACTIONS.

**cypher**

CALL {

*// query*

} IN TRANSACTIONS [OF X ROWS]



For example, the following query would create the Person nodes in individual transactions.

**cypher**

CALL {

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/persons.csv'

AS row

MERGE (p:Person {tmdbId: toInteger(row.person\_tmdbId)})

SET

p.imdbId = toInteger(row.person\_imdbId),

p.bornIn = row.bornIn,

p.name = row.name,

p.bio = row.bio,

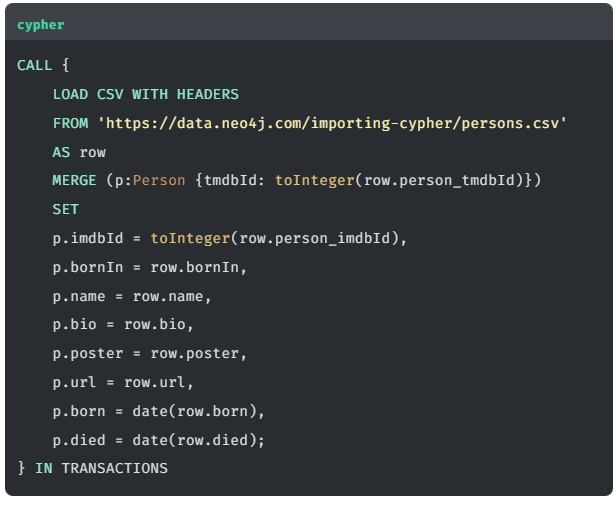
p.poster = row.poster,

p.url = row.url,

p.born = date(row.born),

p.died = date(row.died);

} IN TRANSACTIONS

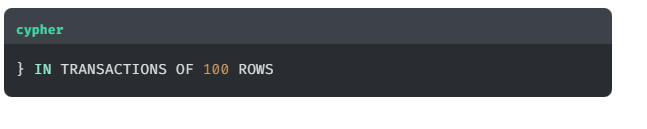


You can batch the transactions by specifying the number of rows to process in each transaction.

For example, modifying the query above to process 100 rows in each transaction:

**cypher**

} IN TRANSACTIONS OF 100 ROWS



# Multiple passes

In the examples you have been exploring, a single file represents a single entity in the graph - persons.csv contained Person nodes and movies.csv contained Movie nodes.

In this lesson, you will explore a single file that stores multiple nodes and relationships and the challenges of importing it into the graph.

Here is the book data you reviewed in a previous lesson:

**csv**

Copy to Clipboard

id,title,author,publication\_year,genre,rating,still\_in\_print,last\_purchased

19515,The Heights,Anne Conrad,2012,Comedy,5,true,2023/4/12 8:17:00

39913,Starship Ghost,Michael Tyler,1985,Science Fiction|Horror,4.2,false,2022/01/16 17:15:56

60980,The Death Proxy,Tim Brown,2002,Horror,2.1,true,2023/11/26 8:34:26

18793,Chocolate Timeline,Mary R. Robb,1924,Romance,3.5,false,2022/9/17 14:23:45

67162,Stories of Three,Eleanor Link,2022,Romance|Comedy,2,true,2023/03/12 16:01:23

25987,Route Down Below,Tim Brown,2006,Horror,4.1,true,2023/09/24 15:34:18

The data could be modeled in a graph as follows:



The data is relatively simple, and a single Cypher query could import it into the graph:

**cypher**

Copy to ClipboardRun in Sandbox

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/books.csv'

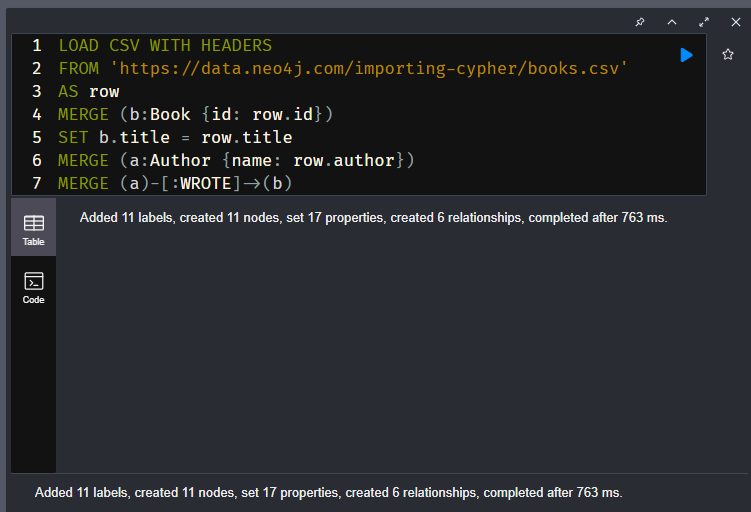
AS row

MERGE (b:Book {id: row.id})

SET b.title = row.title

MERGE (a:Author {name: row.author})

MERGE (a)-[:WROTE]->(b)



Review the query and identify where it creates the Book and Author nodes and WROTE relationship.

However, if this was a more complicated data set with significantly more rows, you may experience issues with the import as it creates related data in a single pass.

Queries with multiple operations chained together have the potential to write data and then read data that is out of sync - which can result in an **Eager** operator.

The Eager operator will cause any operations to execute in their entirety before continuing, ensuring isolation between the different parts of the query. When importing data the Eager operator can cause high memory usage and performance issues.

A mechanism for avoiding the Eager operator is to break the import into smaller parts. By taking multiple passes over the data file, the query also becomes simpler to understand and change to fit the data model.

In this example, the import could be broken into three parts:

* Create Book nodes
* Create Author nodes
* Create WROTE relationships

**cypher**

Copy to ClipboardRun in Sandbox

*// Create `Book` nodes*

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/books.csv'

AS row

MERGE (b:Book {id: row.id})

SET b.title = row.title;

*// Create `Author` nodes*

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing-cypher/books.csv'

AS row

MERGE (a:Author {name: row.author});

*// Create `WROTE` relationships*

LOAD CSV WITH HEADERS

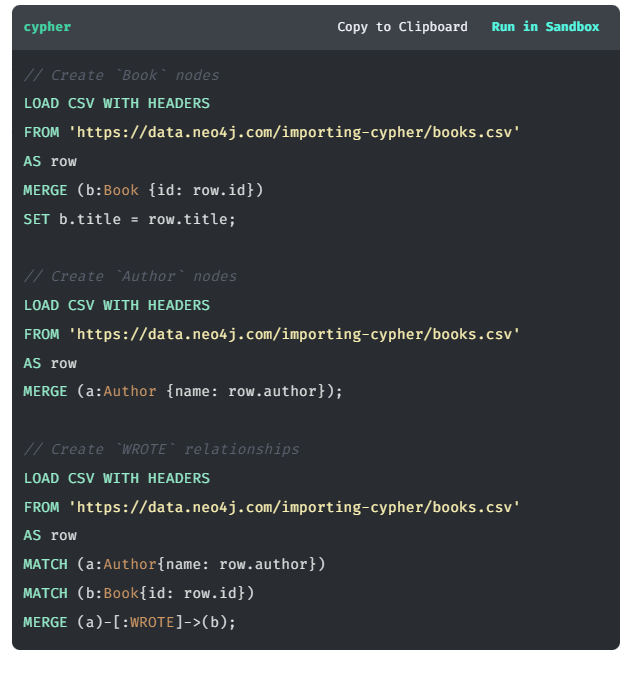
FROM 'https://data.neo4j.com/importing-cypher/books.csv'

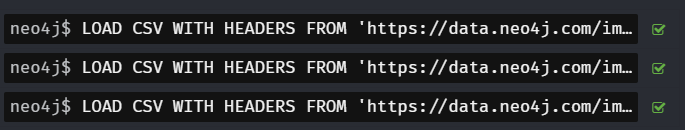
AS row

MATCH (a:Author{name: row.author})

MATCH (b:Book{id: row.id})

MERGE (a)-[:WROTE]->(b);





You can read more about the [**Eager operator in the Neo4j documentation**](https://neo4j.com/docs/cypher-manual/current/execution-plans/operators/#query-plan-eager), how to avoid it while importing data in the [**Avoiding the Eager Operator**](https://www.markhneedham.com/blog/2014/10/23/neo4j-cypher-avoiding-the-eager/) blog post, and [**Cypher Sleuthing: the eager operator**](https://medium.com/neo4j/cypher-sleuthing-the-eager-operator-84a64d91a452) for a detailed walk-through.

Main Point: A mechanism for avoiding the Eager operator is to break the import into smaller parts.

# Other Import Options

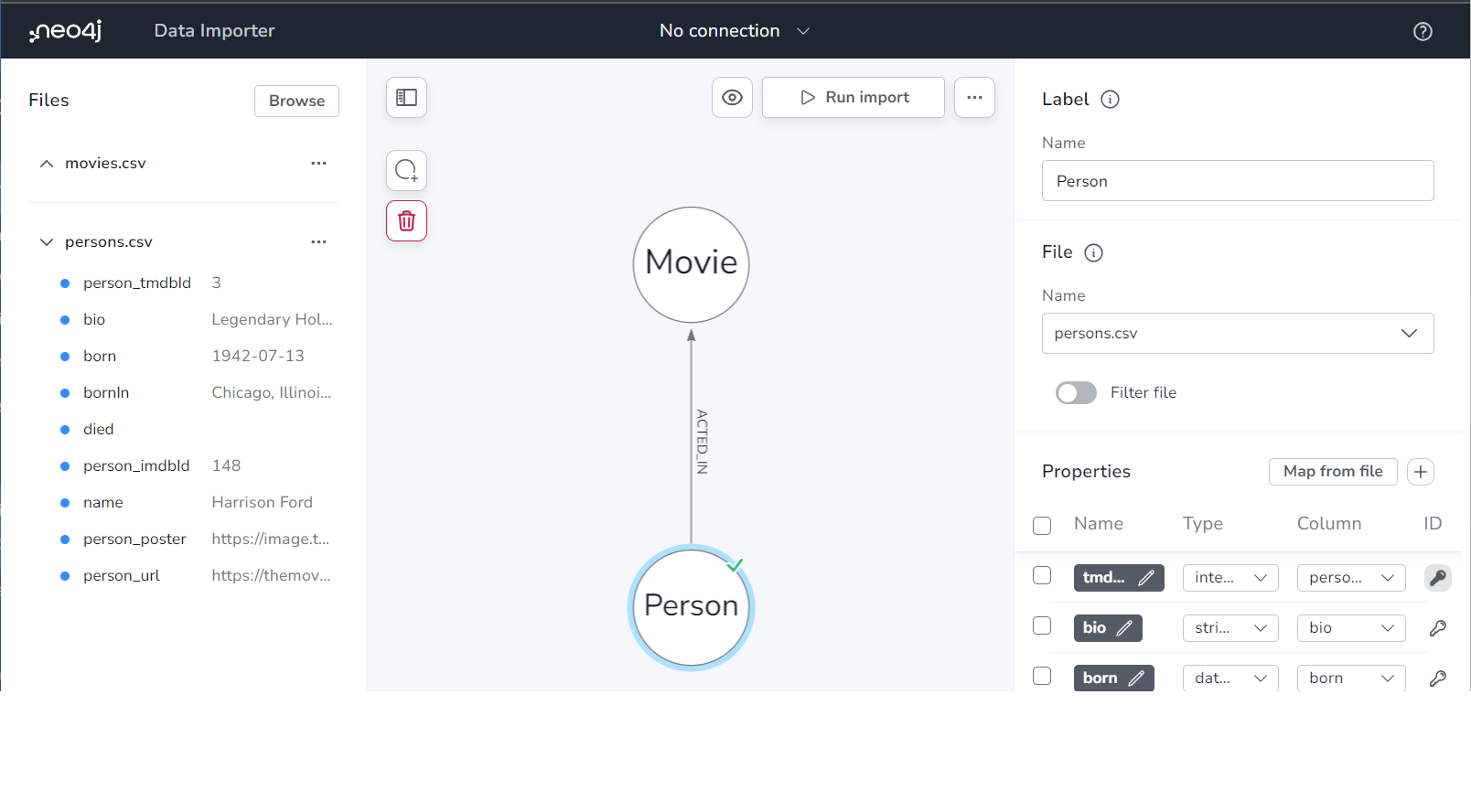
LOAD CSV is a convenient tool for importing data into Neo4j; there are however some considerations:

* You may need to write Cypher statements to transform the data into the desired format. Depending on the data complexity, this can be time-consuming.
* LOAD CSV may not be suitable for large data sets. The definition of large data sets depends on multiple factors, including the data structure, the use case, and the environment. A guideline of over 10 million rows is a good indicator for considering other import options.
* You may need to import data from multiple sources, and LOAD CSV may not be the best option for all of them.
* Complex business rules may require integration to other systems or data sources.

There are a multitude of options for importing data into Neo4j. The following sections provide an overview of some of the possibilities.

## **Neo4j Data Importer**

The [**Neo4j Data Importer**](https://data-importer.neo4j.io/) is a UI-based data importing tool. The data importer is ideal to get started quickly with testing and prototyping. However, there are limited options for data transformation, and it is unsuitable for large data sets.



## neo4j-admin

The [neo4j-admin import**command line interface**](https://neo4j.com/docs/operations-manual/current/tools/neo4j-admin/neo4j-admin-import/) supports importing large data sets. neo4j-admin import converts CSV files into the internal binary format of Neo4j and can import millions of rows within minutes.

You need to structure the import files in a specific way, and the database needs to be offline when the import is run.

## ETL (Extract, Transform, Load) Tool

An ETL tool, for example [**Apache Hop**](https://hop.apache.org/), is a good choice for importing data from multiple sources. ETL tools generally support various data sources, can transform data into the desired format, and have visualization tools.

## Custom integration using Neo4j drivers

Building a custom application to load data into the graph database is a good option if you have complex business rules or need to integrate with other systems. There are several **[GraphAcademy courses for developers](https://graphacademy.neo4j.com/categories/developer/" \t "_blank)** where you can learn how to build applications using Neo4j drivers.

# Course Summary

Congratulations on completing "Import CSV data into Neo4j".

You have learned how to:

* Import CSV data into Neo4j using Cypher.
* Use the MERGE clause to create nodes and relationships from the CSV data.
* Cast properties to the correct data type.
* Create constraints on properties.
* How to manage larger data imports.

Continue your learning with the following resources:

* [**GraphAcademy**](https://graphacademy.neo4j.com/) - Free online training for Neo4j
* [**Importing CSV data into Neo4j**](https://neo4j.com/docs/getting-started/data-import/csv-import/) - The Neo4j Getting Started guide