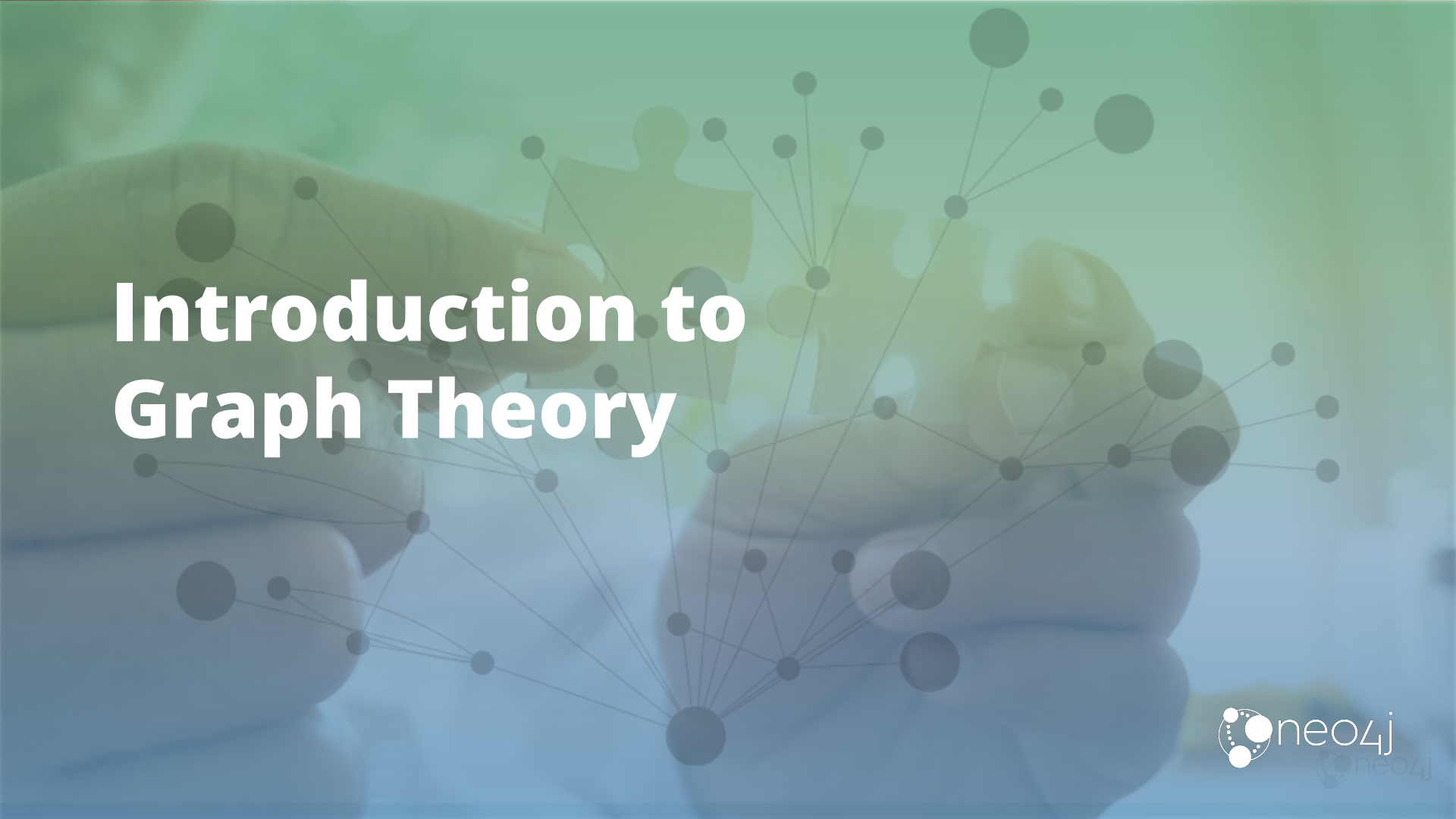
# Introduction to Neo4j

V 1.1

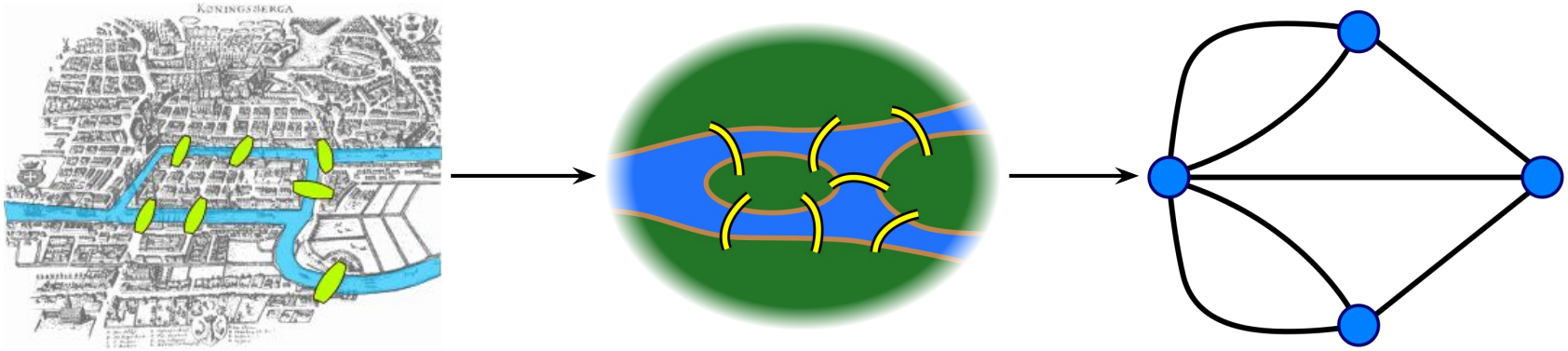
neo4j

# Introduction to Graph Theory



# What is a Graph?

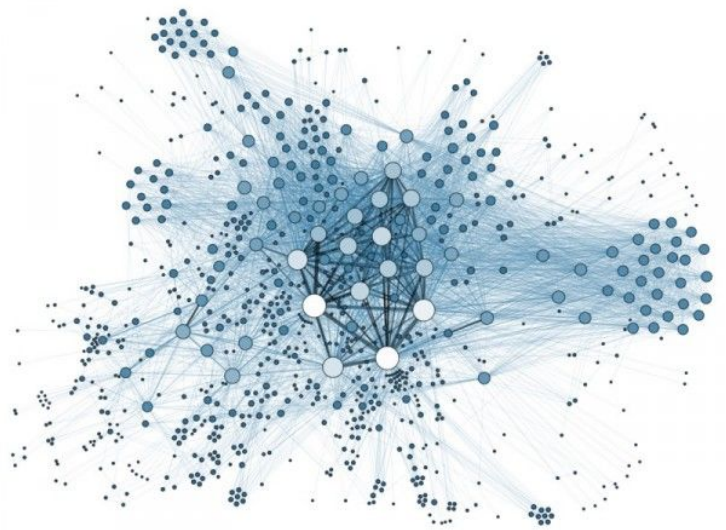
A **graph** is set of discrete objects, each of which has some set of relationships with the other objects



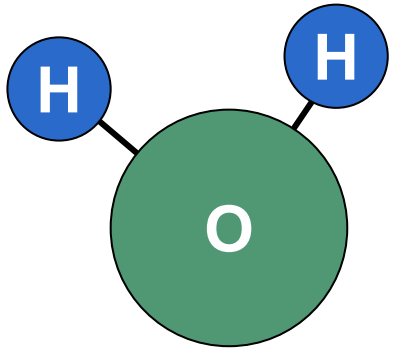

*Seven Bridges of Konigsberg problem. Leonhard Euler, 1735*

# Anything can be a graph

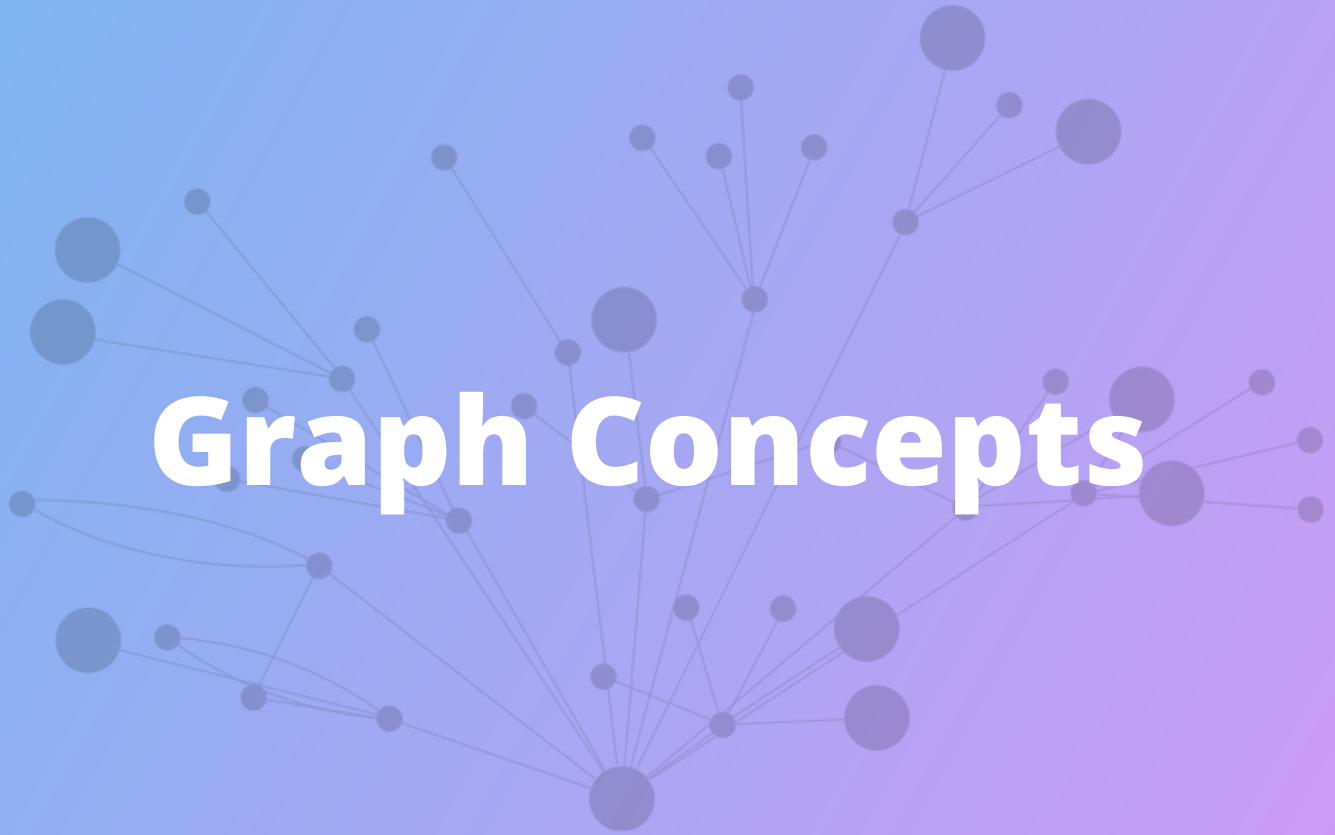
the Internet



a water molecule



# Graph Concepts

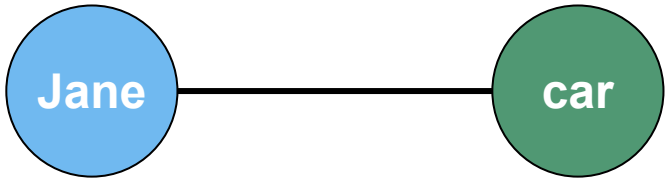


# Graph Components

**Node** (Vertex)

- The main data element from which graphs are constructed
  *A node without relationships is permitted.  A relationship without nodes is not.*
- A waypoint along a traversal route

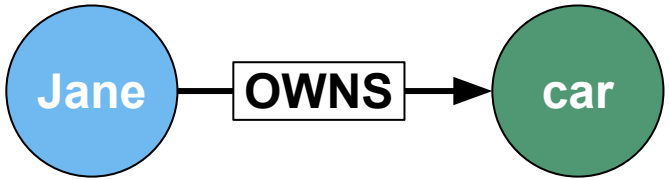
**Relationship** (Edge)



# Graph Components

## **Node** (Vertex)

- The main data element from which graphs are constructed
  *A node without relationships is permitted.  A relationship without nodes is not.*
- A waypoint along a traversal route

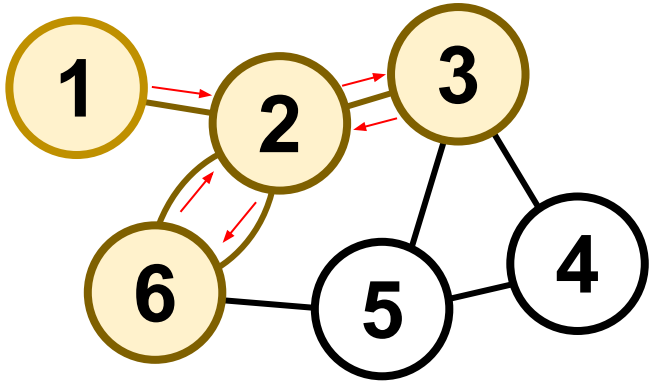



## **Relationship** (Edge)

- A link between two nodes.  May contain:
  - Direction
  - Metadata; e.g. *weight* or *relationship type*

# Traversal

- **Walk**: an ordered, alternating sequence of nodes and relationships
  *Nodes and relationships can be repeated*

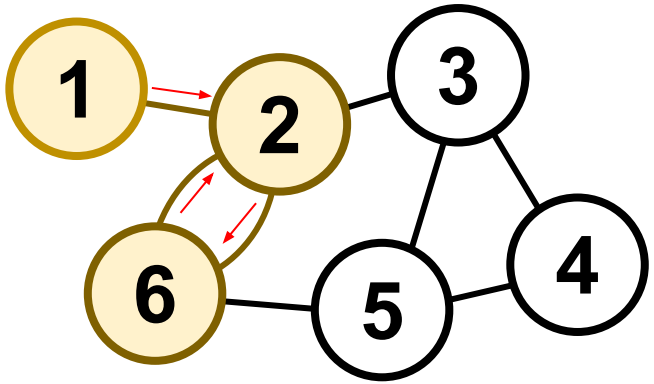



**1-2-3-2-6-2**

# Traversal

- **Walk**: an ordered, alternating sequence of nodes and relationships
  *Nodes and relationships can be repeated*

- **Trail**: A walk in which no relationships are repeated
  *Nodes can be repeated*




1-2-3-2-6-2

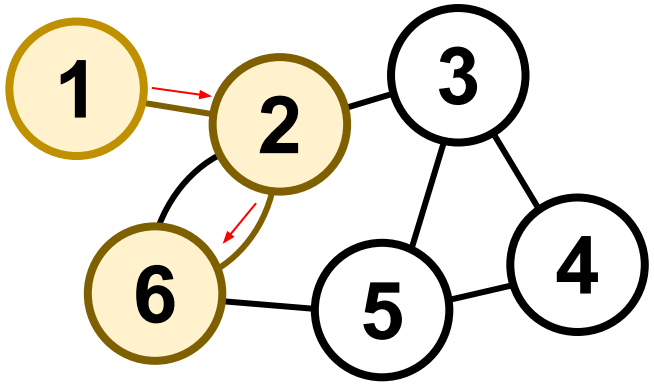
**1-2-6-2**

# Traversal

- **Walk**: an ordered, alternating sequence of nodes and relationships
  *Nodes and relationships can be repeated*

- **Trail**: A walk in which no relationships are repeated
  *Nodes can be repeated*

- **Path**: A trail in which no nodes are repeated
  *I.e., a walk where all items are unique*




1-2-3-2-6-2

1-2-6-2

**1-2-6**

# Summary

# Summary

- A **graph** is a data model consisting of a set of **nodes** linked by **relationships**

- The purpose of modeling data as a graph is to enable **traversal**

- There are multiple traversal types, the most important of which is a **path**
  *an ordered, alternating set of nodes and relationships*
  *in which all items are unique*

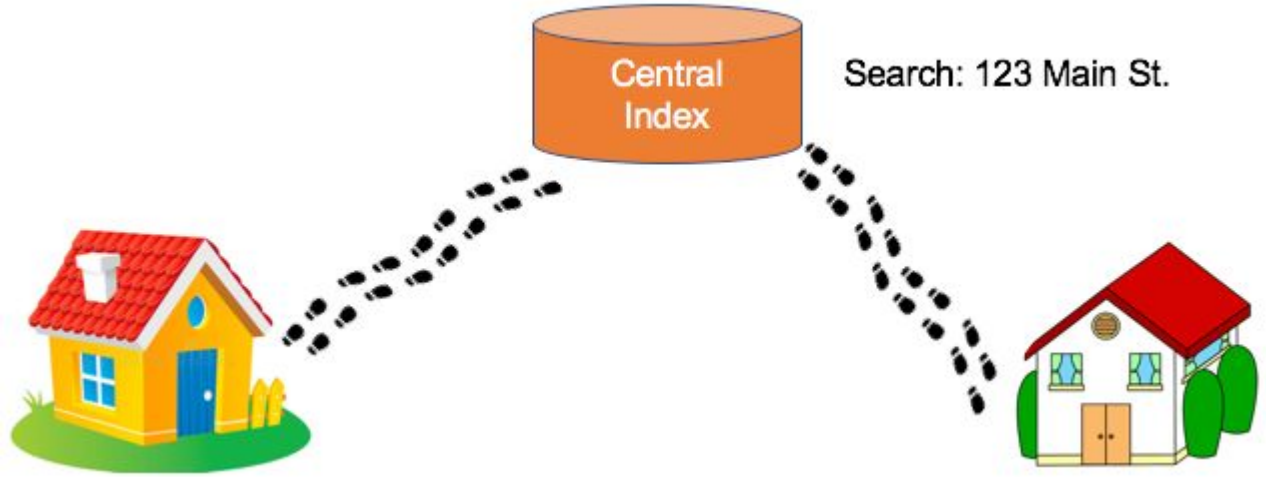
# Graph Databases & Index-Free Adjacency

# IFA in 2 Minutes

My house

Anne’s house

# IFA in 2 Minutes




My house

Anne’s house

# The Relational Data Problem

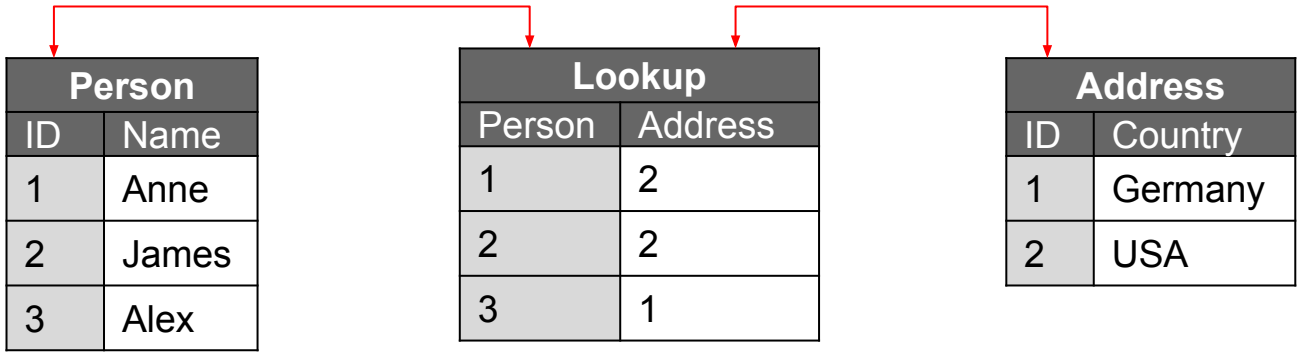



neo4j

# Relationships in RDBMS

- Require foreign keys, and possibly a lookup table
- Traversing a foreign key requires an index lookup

*The purpose of graphs is to do rapid traversal. The RDBMS model is too expensive for that.*



| Person | |
|---|---|
| ID | Name |
| 1 | Anne |
| 2 | James |
| 3 | Alex |

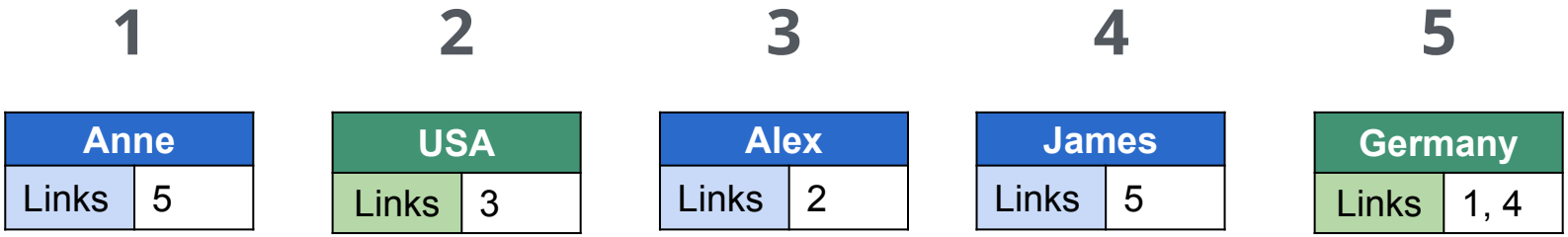
| Lookup | |
|---|---|
| Person | Address |
| 1 | 2 |
| 2 | 2 |
| 3 | 1 |

| Address | |
|---|---|
| ID | Country |
| 1 | Germany |
| 2 | USA |

# Index-Free Adjacency

# IFA Benefits

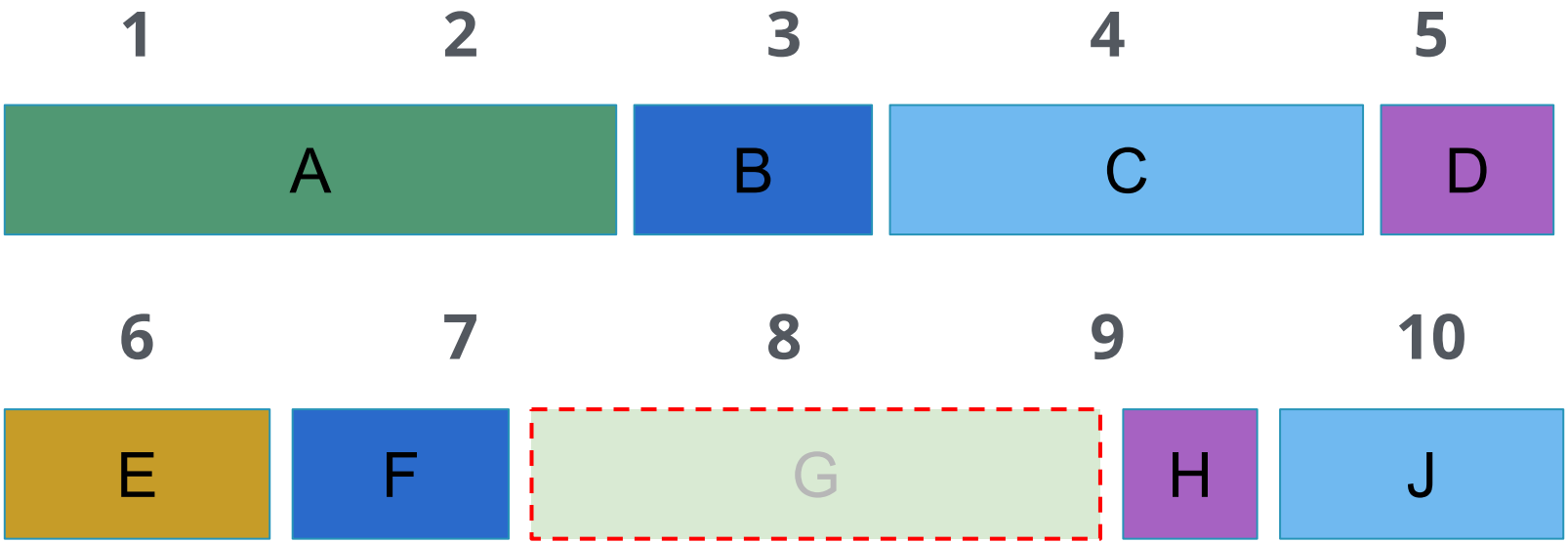
IFA uses pointers to the target address




- *No index lookups or table scans*
- *Reduced duplication of foreign keys*

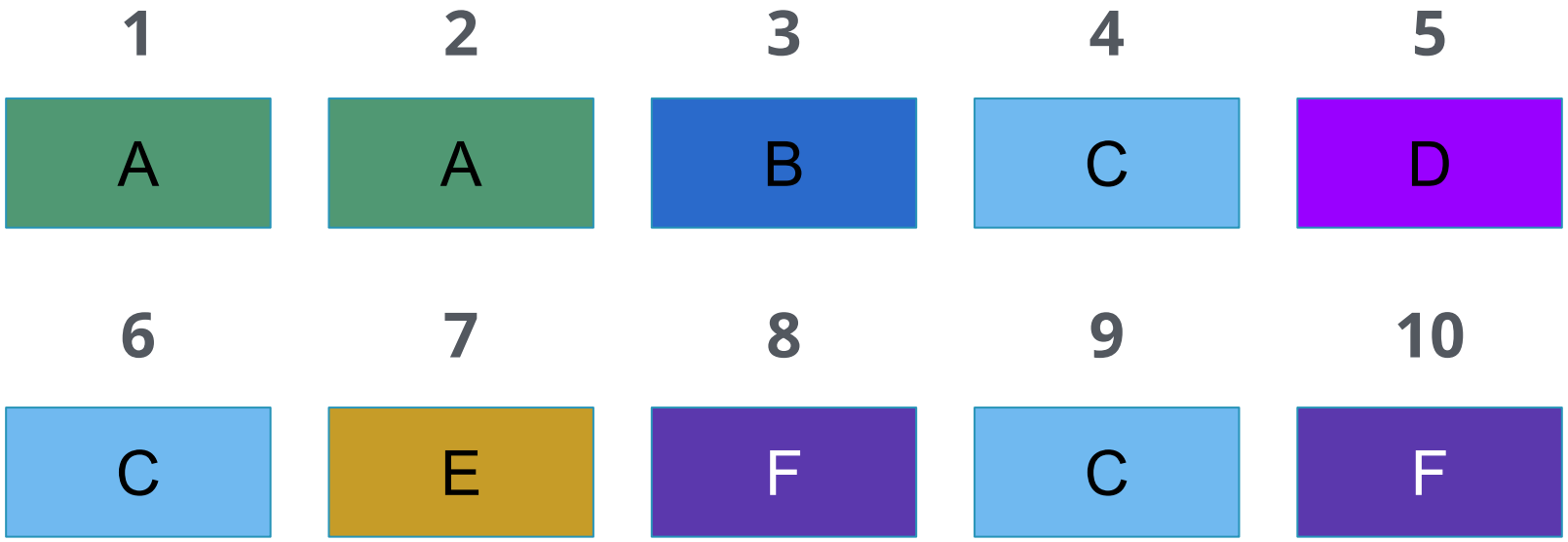
# IFA Question: Deletion

How to avoid wasted space on deletion and insertion?



# IFA Solution: Chunking

All data objects are chunked into same-size pieces



# IFA Solution: Chunking

- Related chunks need not be adjacent
  - *e.g., a four-chunk object can use the first four open locations, no matter where they are*
- Related chunks reference one another via IFA

***Effect:*** scanning for a value within a large data item can involve many "hidden" hops

***Corollary:*** it is frequently cheaper to perform multiple relationship hops rather than scan for a value within a single large object

***Corollary****:* in Neo4j, every object should store as little data as is feasible

# Summary

# Summary

- RDBMS is very inefficient at traversal
  - *Foreign keys, lookup tables, and a four-step process are required*
- Index-Free Adjacency speeds traversal by using **pointers** instead of keys and lookup tables

- Neo4j uses **chunking** to store data maximally efficiently
  - *Because of this, all objects should store as little data as feasible*

# Introduction to Neo4j

# Overview

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

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

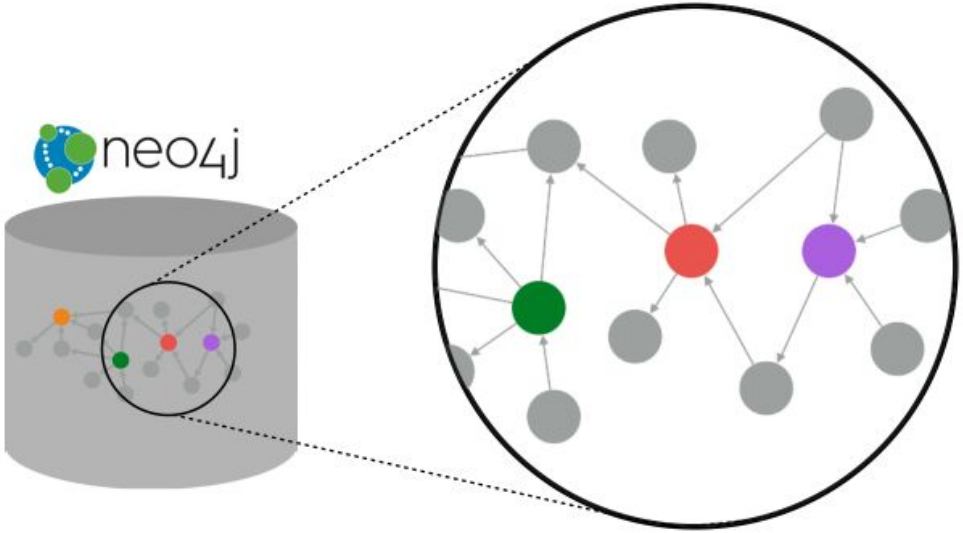
# Neo4j Graph Platform

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

The heart of the Neo4j Graph Platform is the Neo4j Database.

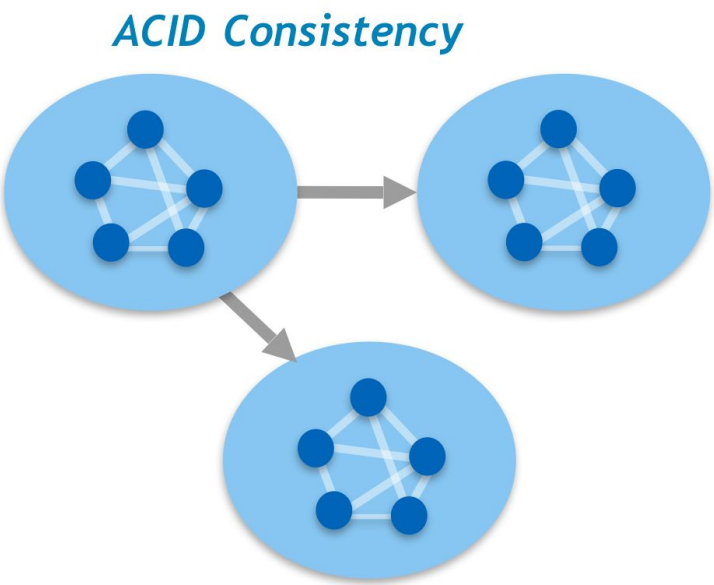
# Neo4j Database: Index-free adjacency

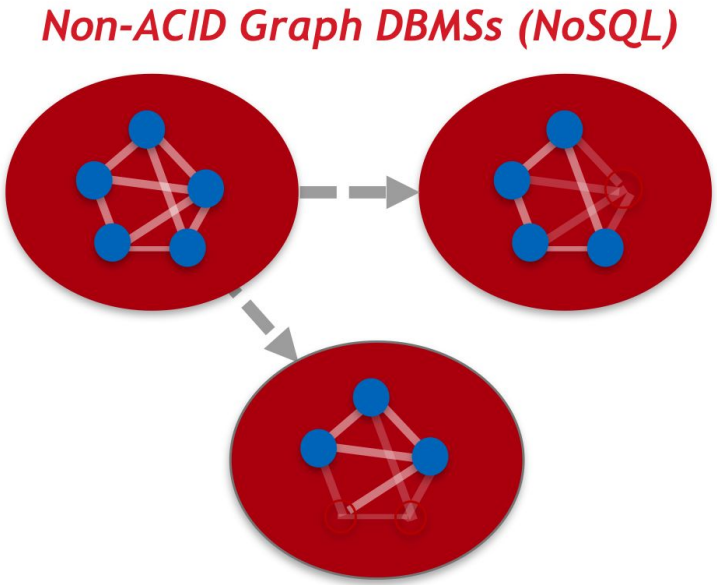
Nodes and relationships are stored on disk as a graph for fast navigational access using pointers.



# Neo4j Database: ACID

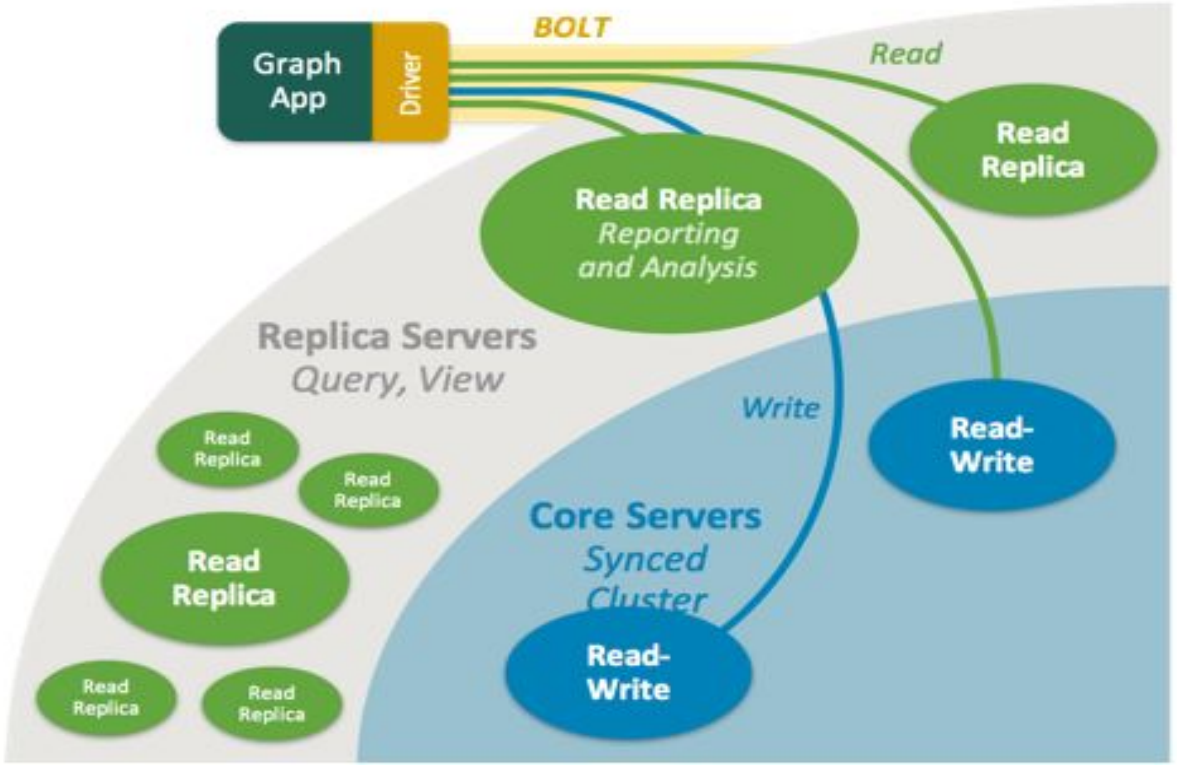
Transactional consistency - all updates either succeed or fail.



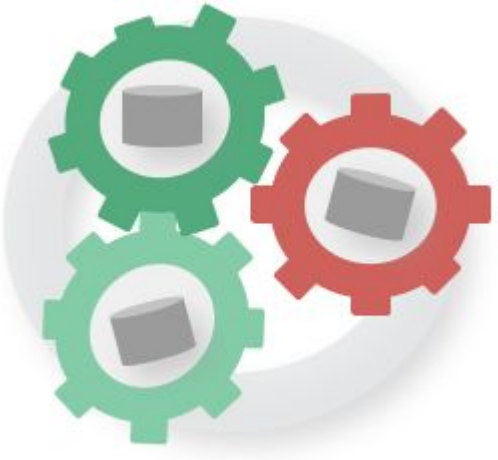
# Clusters

ACID across locations.



# Graph engine

- Interpret Cypher statements
- Store and retrieve data
- Kernel-level access to filesystem
- Scalable
- Performant



# Language and driver support

- Cypher to access the database
- Open source Cypher
- You can write server-side extensions to access the database
- Out-of-the-box drivers to access the database via **bolt** protocol:
  - Java
  - JavaScript
  - Python
  - C#
  - Go
- Neo4j community contributions for other languages

# Libraries

Out-of-the-box:

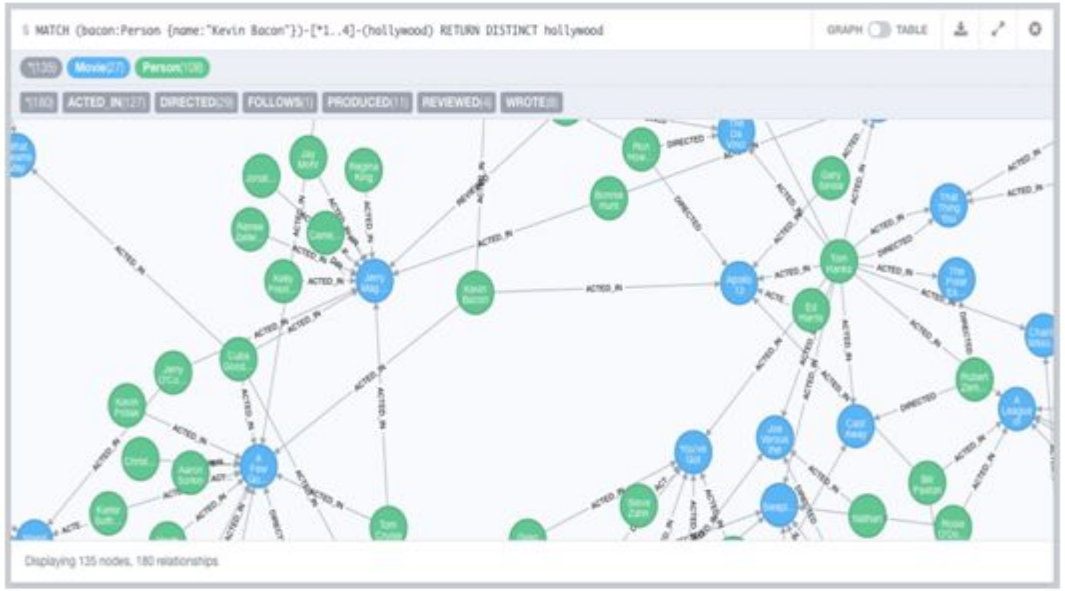
- Awesome Procedures on Cypher (APOC)
- Graph Algorithms
- GraphQL

Neo4j community has contributed many specialized libraries also.

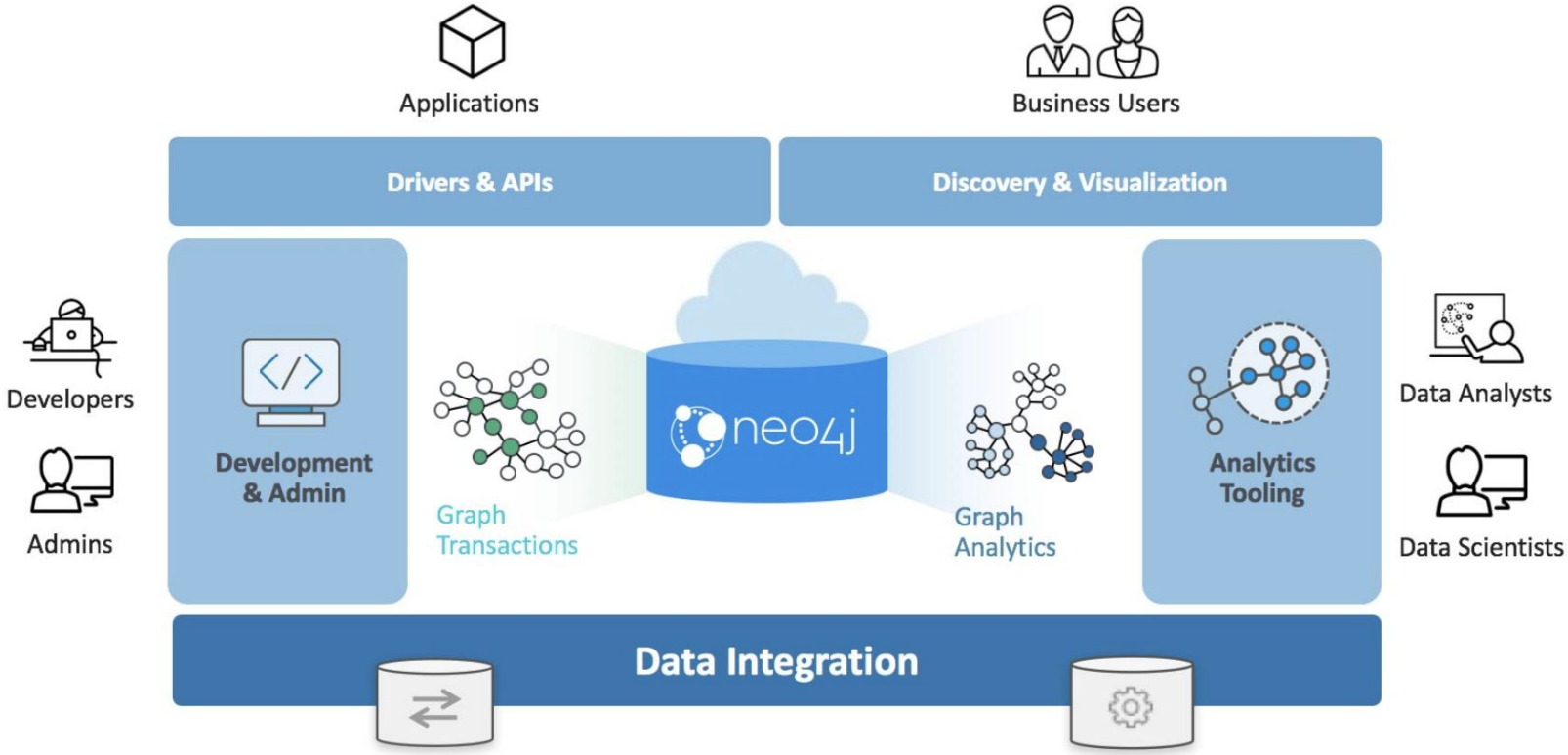
# Tools

- Neo4j Desktop
- Neo4j Browser
- Neo4j Bloom
- Neo4j ETL Tool

Neo4j community has contributed many specialized tools also.



# Neo4j Graph Platform architecture




Check your understanding

# Question 1

What are some of the benefits provided by the Neo4j Graph Platform?

Select the correct answers.

- ❑ Database clustering
- ❑ ACID
- ❑ Index free adjacency
- ❑ Optimized graph engine

# Answer 1

What are some of the benefits provided by the Neo4j Graph Platform?

Select the correct answers.

✔️ Database clustering

✔️ ACID

✔️ Index free adjacency

✔️ Optimized graph engine

# Question 2

What libraries are included with Neo4j Graph Platform?

Select the correct answers.

- ❑ APOC
- ❑ JGraph
- ❑ GRAPH ALGORITHMS
- ❑ GraphQL

# Answer 2

What libraries are included with Neo4j Graph Platform?

Select the correct answers.

✔ APOC

❑ JGraph

✔ GRAPH ALGORITHMS

✔ GraphQL

# Question 3

What are some of the language drivers that come with Neo4j out of the box?

Select the correct answers.

- ❑ Java
- ❑ Ruby
- ❑ Python
- ❑ JavaScript

# Answer 3

What are some of the language drivers that come with Neo4j out of the box?

Select the correct answers.

✔ Java

❑ Ruby

✔ Python

✔ JavaScript

# Summary

You should be able to:

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

# Setting Up Your Development Environment

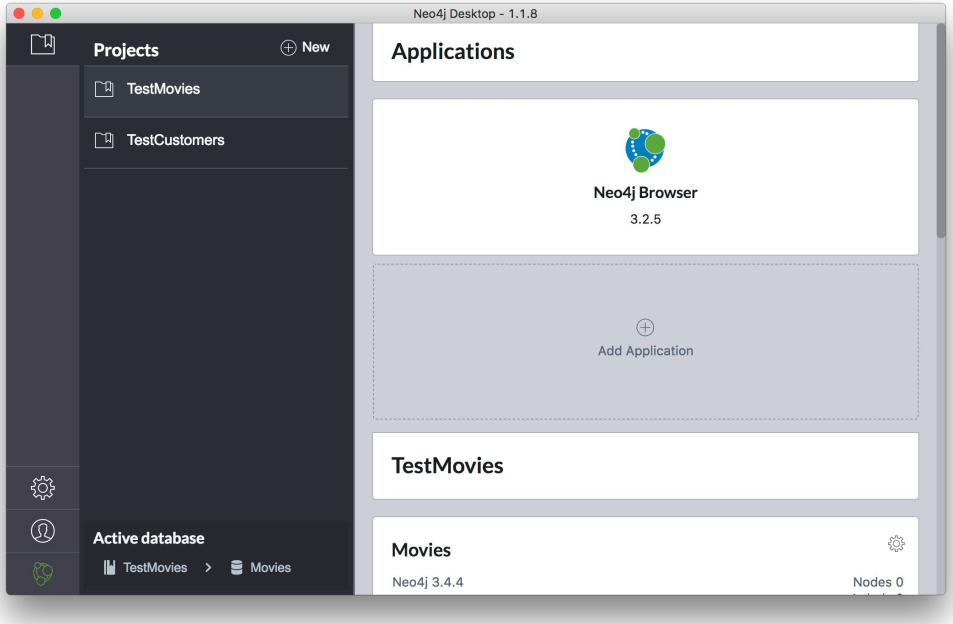
# Overview

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

- Determine your ideal Neo4j development environment
- Start using Neo4j Desktop / Neo4j Sandbox
- Start using Neo4j Browser

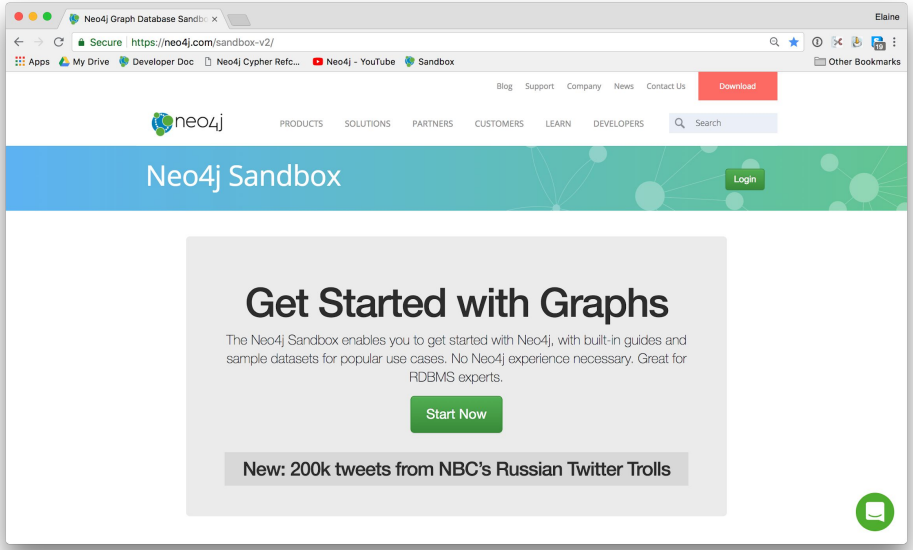
# Neo4j Desktop

- Create local databases
- Manage multiple projects
- Manage Database Server
- Start Neo4j Browser instances
- Install plugins (libraries) for use with a project
- OS X, Linux, Windows



# Neo4j Sandbox

- Web browser access to Neo4j Database Server and Neo4j Database in the cloud
- Comes with a blank or pre-populated database
- Temporary access
- Save Cypher scripts for use in other sandboxes or Neo4j projects
- Instance lives for up to ten days
- No need to install Neo4j on your machine






neo4j

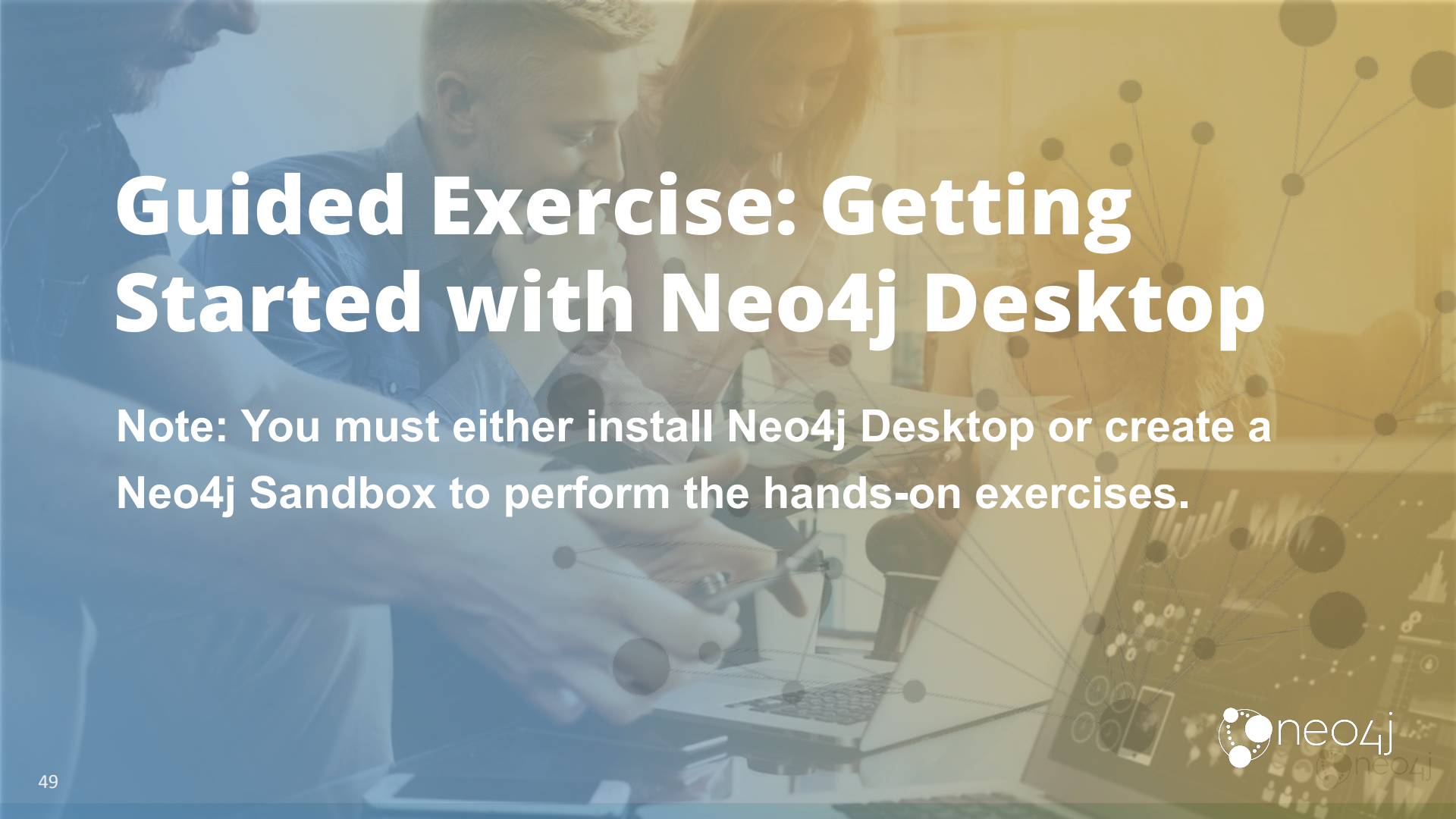
# Setting up your development environment

**If using Neo4j Sandbox:**

1. Start a Neo4j Sandbox (use latest Neo4j release).
   a. Has a blank database that is started.
2. Click the link to access Neo4j Browser.

**If using Neo4j Desktop:**

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

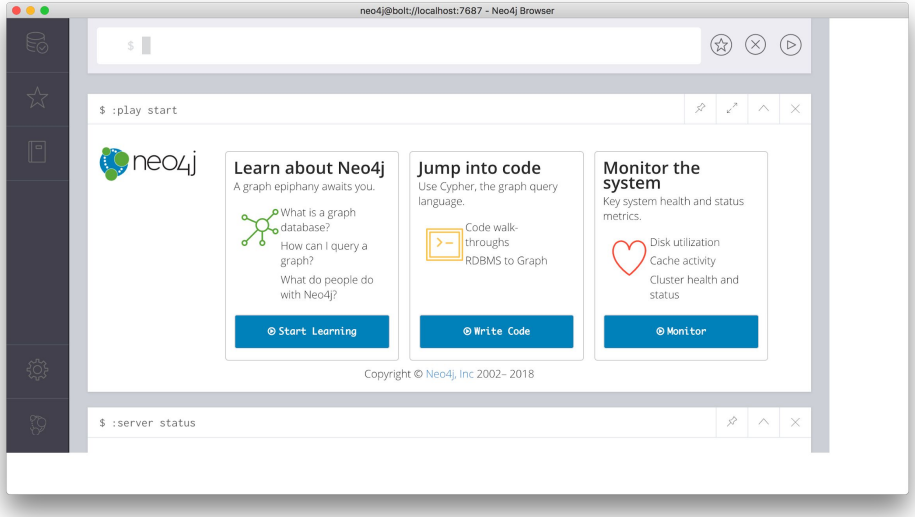


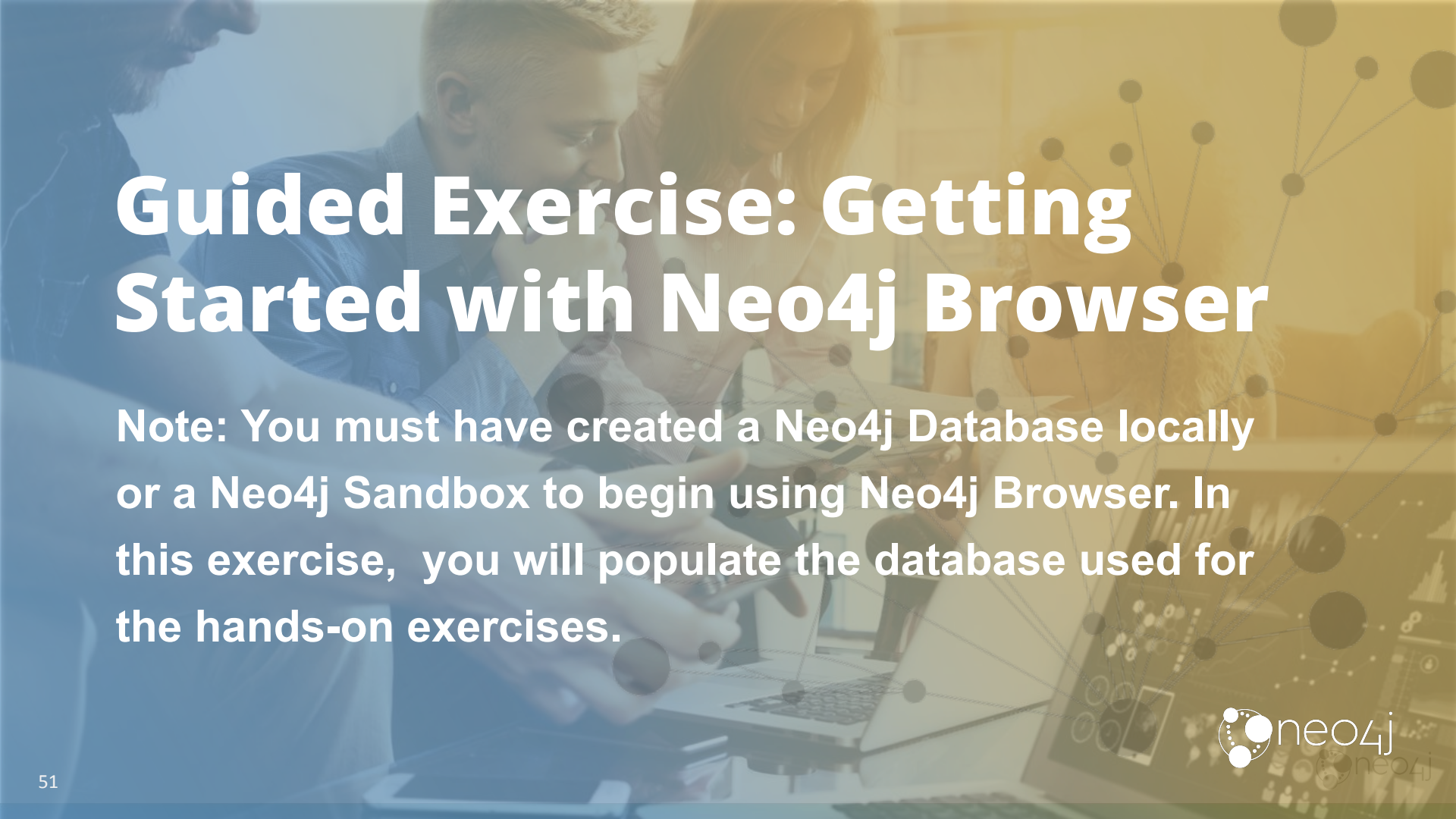
# Guided Exercise: Getting Started with Neo4j Desktop

**Note: You must either install Neo4j Desktop or create a Neo4j Sandbox to perform the hands-on exercises.**

# Neo4j Browser

- Web browser access to Neo4j Database Server and Neo4j Database
- Access local database (Neo4j Desktop) or database in the cloud (Sandbox)
- Access the database with commands or Cypher statements





# Guided Exercise: Getting Started with Neo4j Browser

**Note: You must have created a Neo4j Database locally or a Neo4j Sandbox to begin using Neo4j Browser. In this exercise, you will populate the database used for the hands-on exercises.**

# Check your understanding

# Question 1

What development environment should you use if you want to develop a graph-enabled application using a local Neo4j Database?

Select the correct answer.

- ❏ Neo4j Desktop
- ❏ Neo4j Sandbox

# Answer 1

What development environment should you use if you want to develop a graph-enabled application using a local Neo4j Database?

Select the correct answer.

✔️ Neo4j Desktop

❑ Neo4j Sandbox

# Question 2

What development environment should you use if you want develop a graph-enabled application using a temporary, cloud-based Neo4j Database?

Select the correct answer.

- ❏ Neo4j Desktop
- ❏ Neo4j Sandbox

# Answer 2

What development environment should you use if you want develop a graph-enabled application using a temporary, cloud-based Neo4j Database?

Select the correct answer.

❏ Neo4j Desktop

✔ Neo4j Sandbox

# Question 3

Which Neo4j Browser command do you use to view a browser guide for the Movie graph?

Select the correct answer.

- ❏ MATCH (Movie Graph)
- ❏ :MATCH (Movie Graph)
- ❏ play Movie Graph
- ❏ :play Movie Graph

# Answer 3

Which Neo4j Browser command do you use to view a browser guide for the Movie graph?

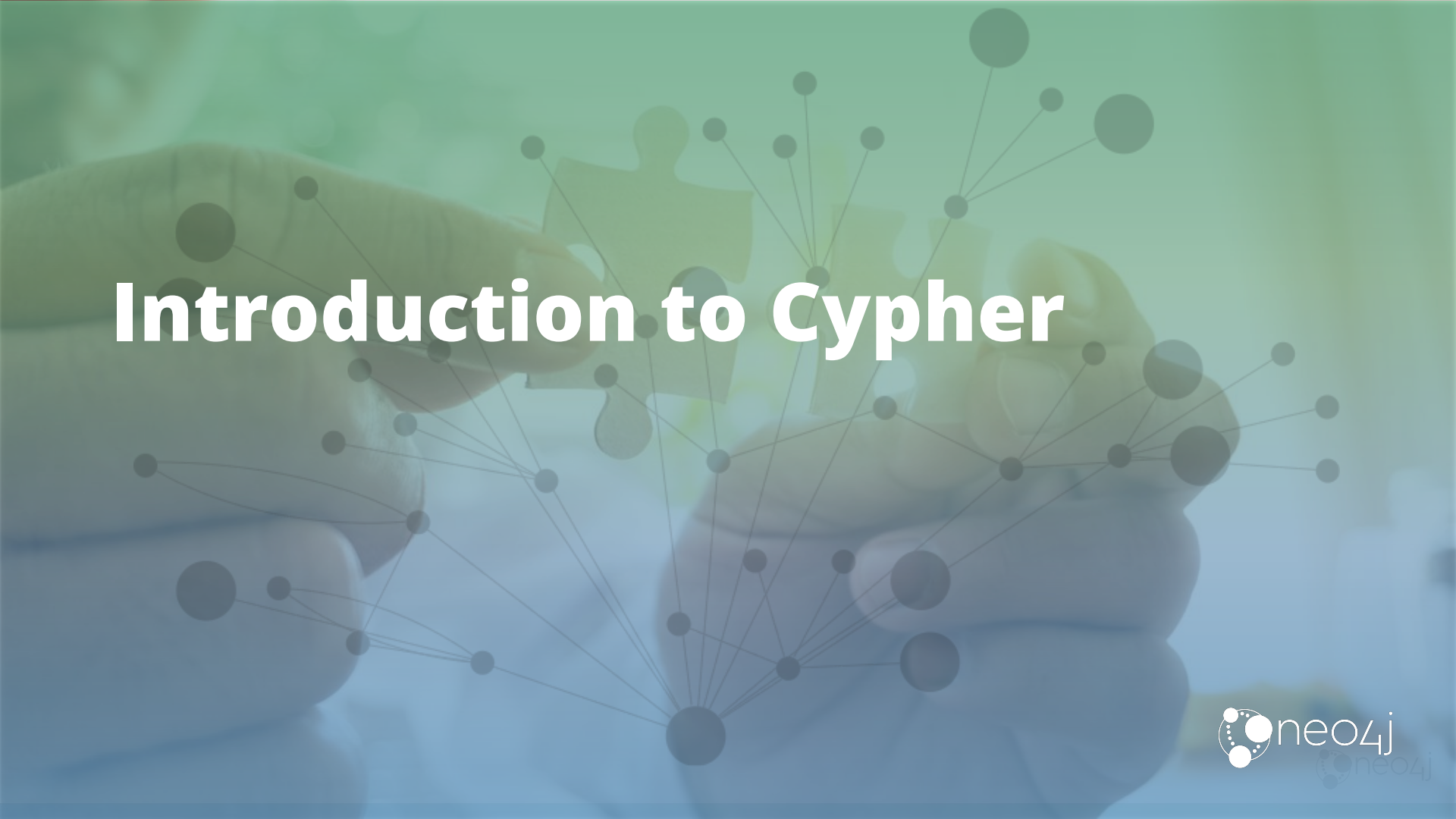
Select the correct answer.

- ❏ MATCH (Movie Graph)
- ❏ :MATCH (Movie Graph)
- ❏ play Movie Graph
- ✔ :play Movie Graph

# Summary

You should be able to:

- Determine your ideal Neo4j development environment
- Start using Neo4j Desktop / Neo4j Sandbox
- Start using Neo4j Browser


Introduction to Cypher
neo4j

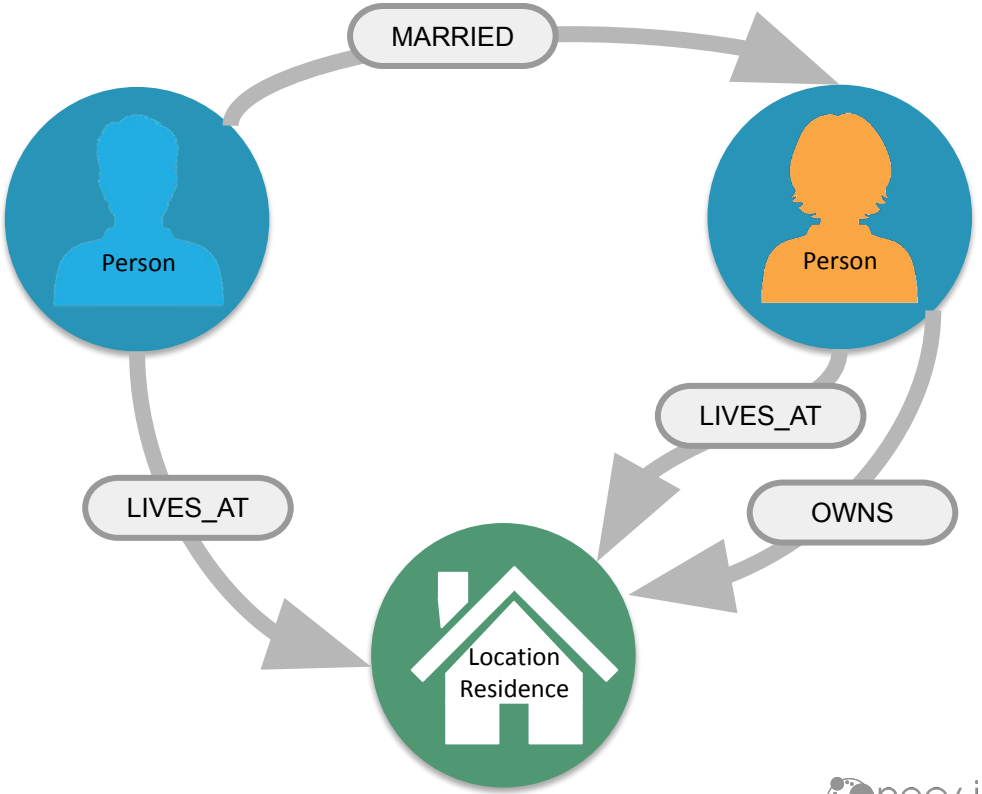
# Overview

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

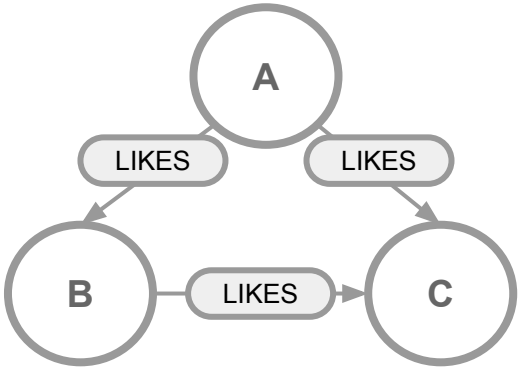
- Retrieve nodes from the graph.
- Filter nodes retrieved using labels and property values of nodes.
- Retrieve property values from nodes in the graph.
- Filter nodes retrieved using relationships.

# What is Cypher?

- Declarative query language
- Focuses on **what**, not how to retrieve
- Uses keywords such as **MATCH**, **WHERE**, **CREATE**
- Runs in the database server for the graph
- ASCII art to represent nodes and relationships

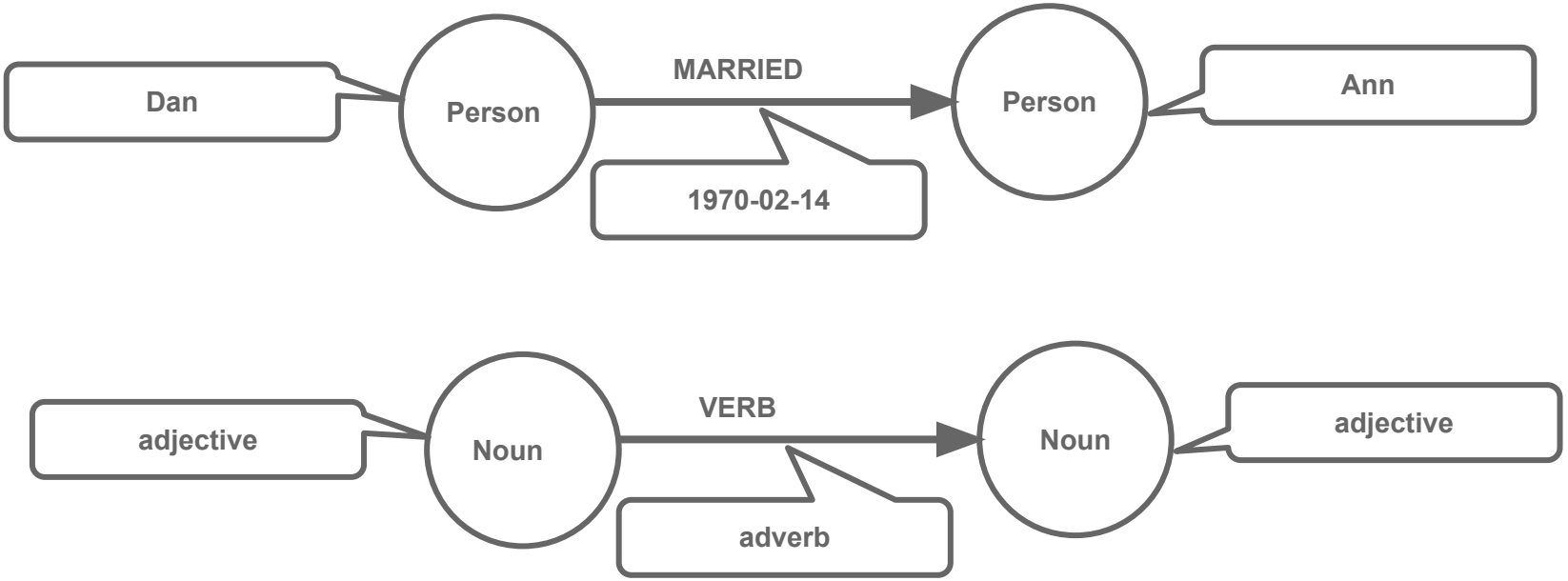


# Cypher is ASCII Art



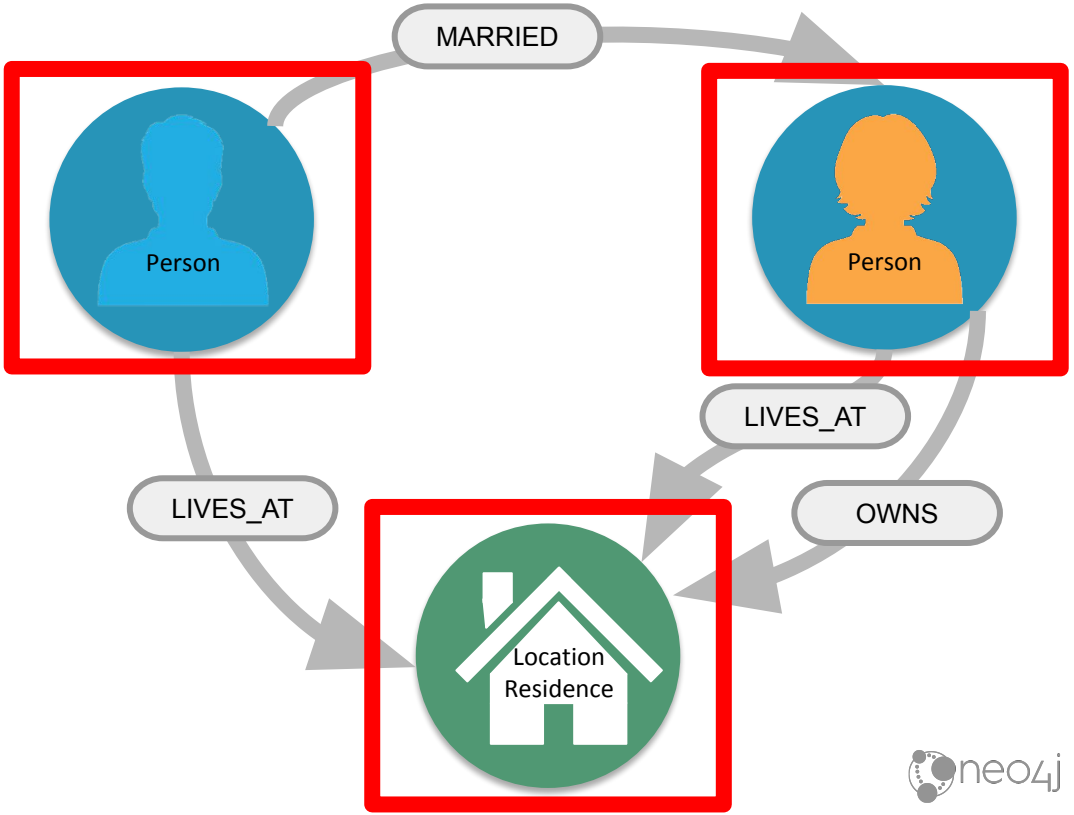

```
(A)-[:LIKES]->(B),(A)-[:LIKES]->(C),(B)-[:LIKES]->(C)

    (A)-[:LIKES]->(B)-[:LIKES]->(C)<-[:LIKES]-(A)
```

# Cypher is readable



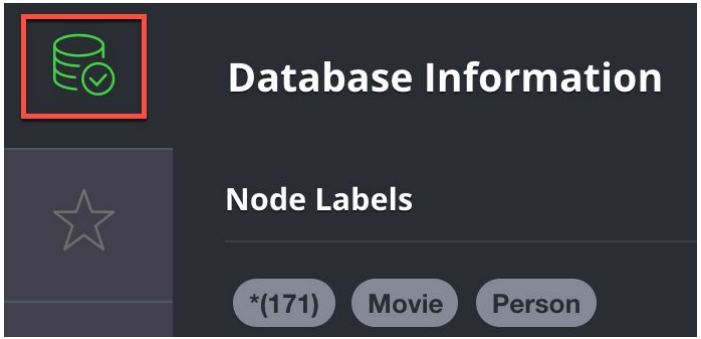
# Nodes

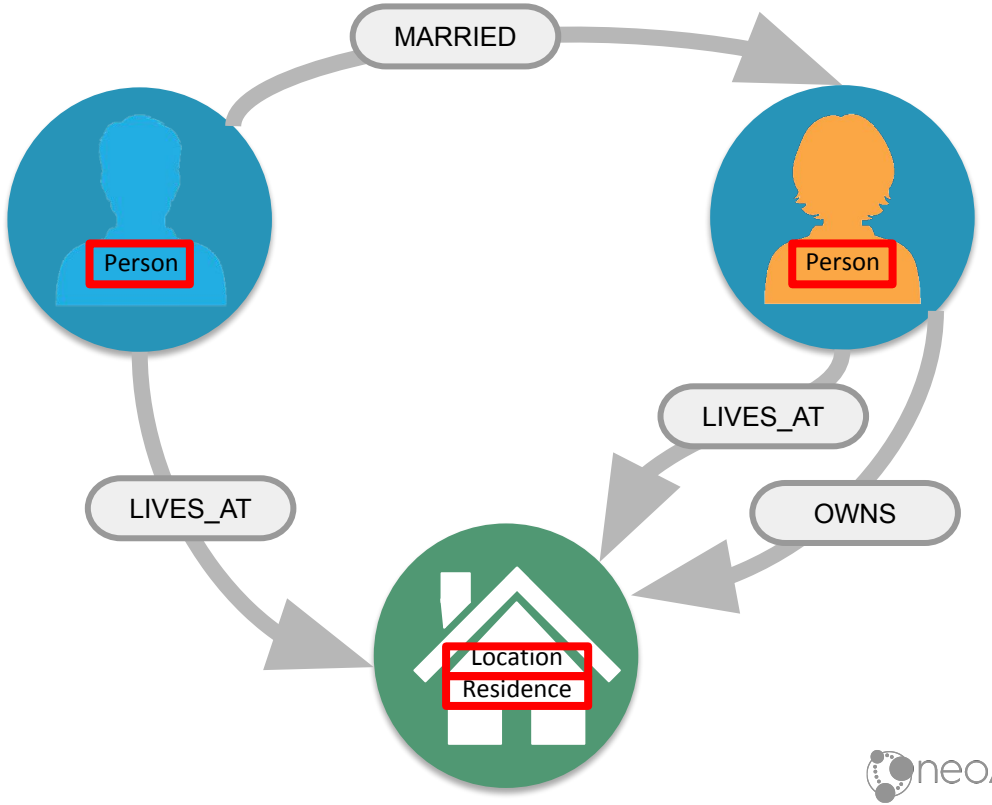
```
()
(p)
(l)
(n)
```



# Labels

```
(:Person)
(p:Person)
(:Location)
(l:Location)
(n:Residence)
(x:Location:Residence)
```

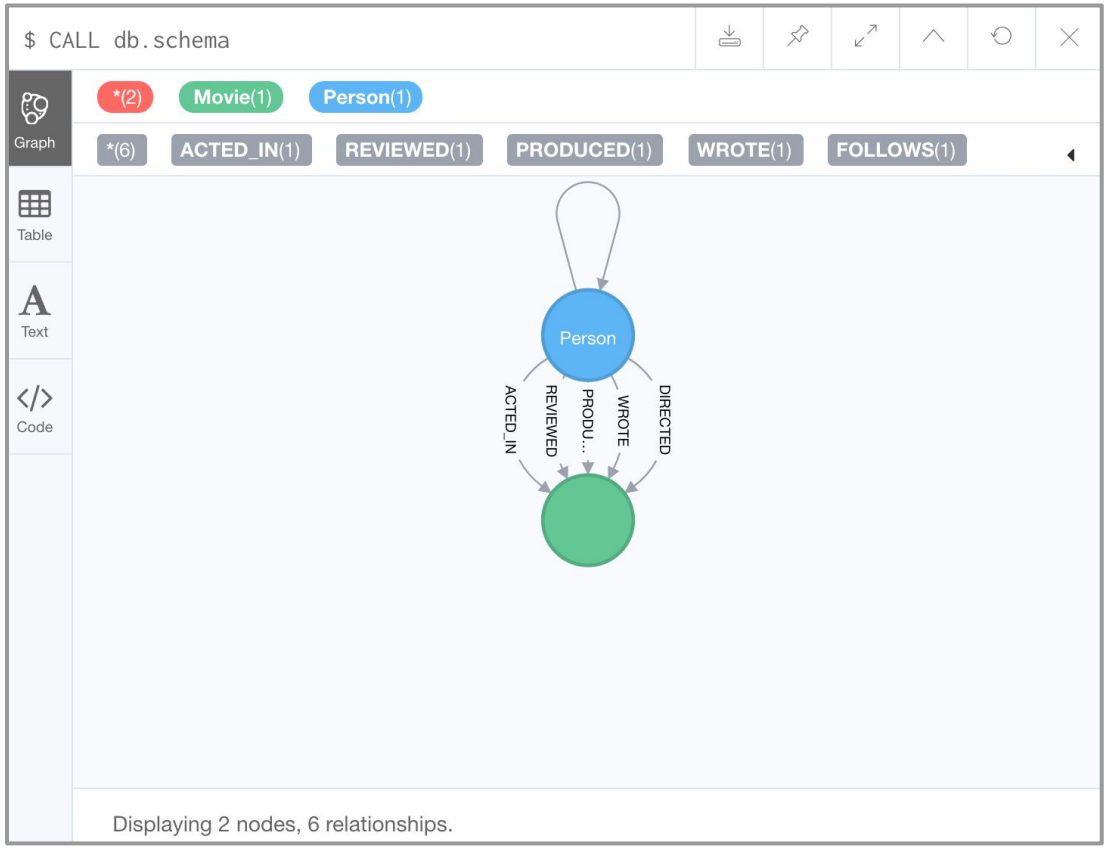


# Comments in Cypher

```
()                  // anonymous node not be referenced later in the query
(p)                 // variable p, a reference to a node used later
(:Person)           // anonymous node of type Person
(p:Person)          // p, a reference to a node of type Person
(p:Actor:Director)  // p, a reference to a node of types Actor and Director
```

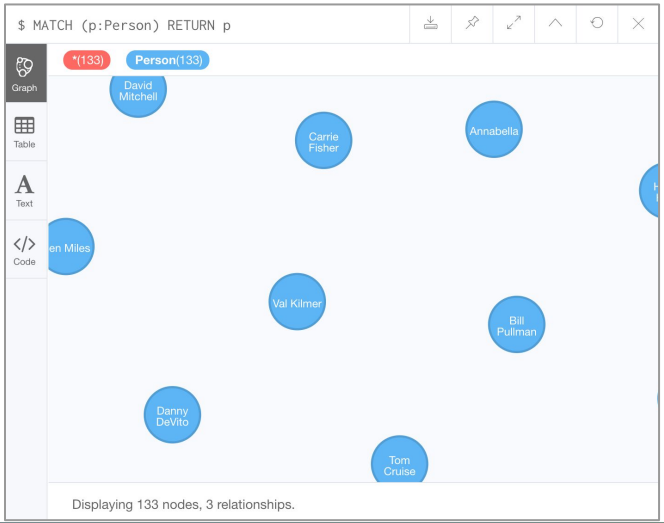
# Examining the data model



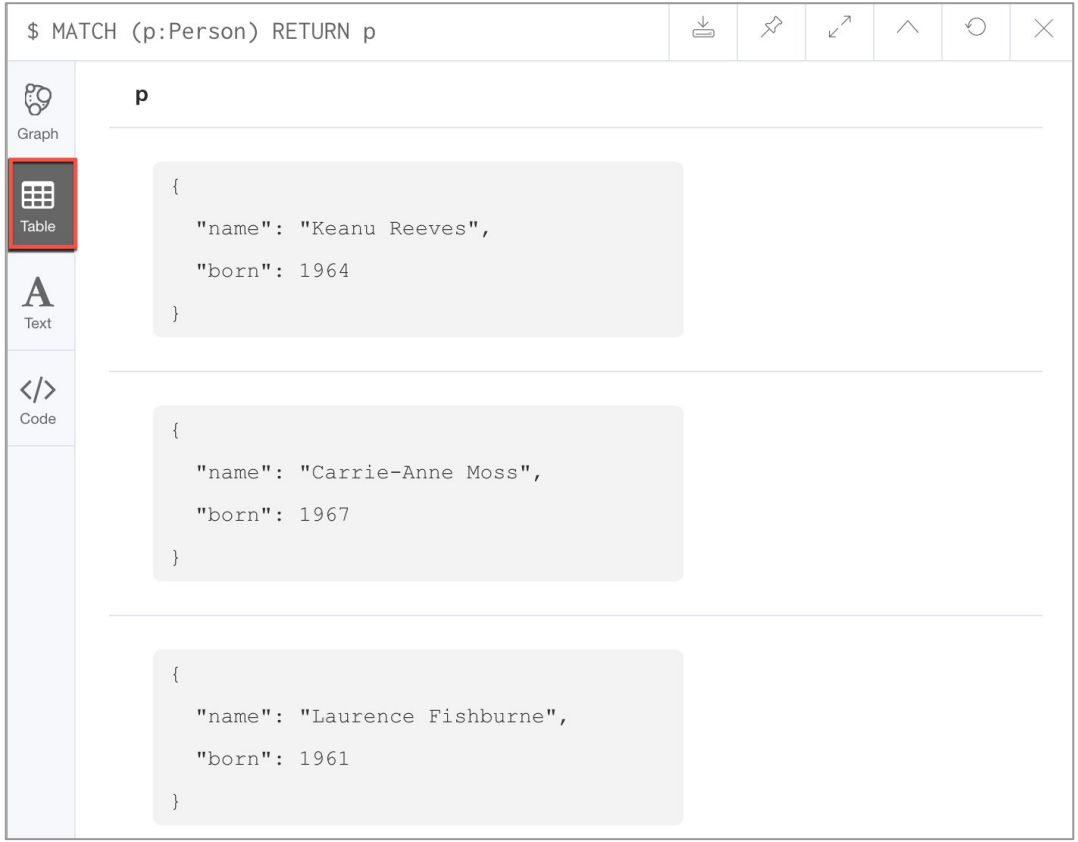
# Using MATCH to retrieve nodes

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

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



# Viewing nodes as table data



# Exercise 1: Retrieving Nodes

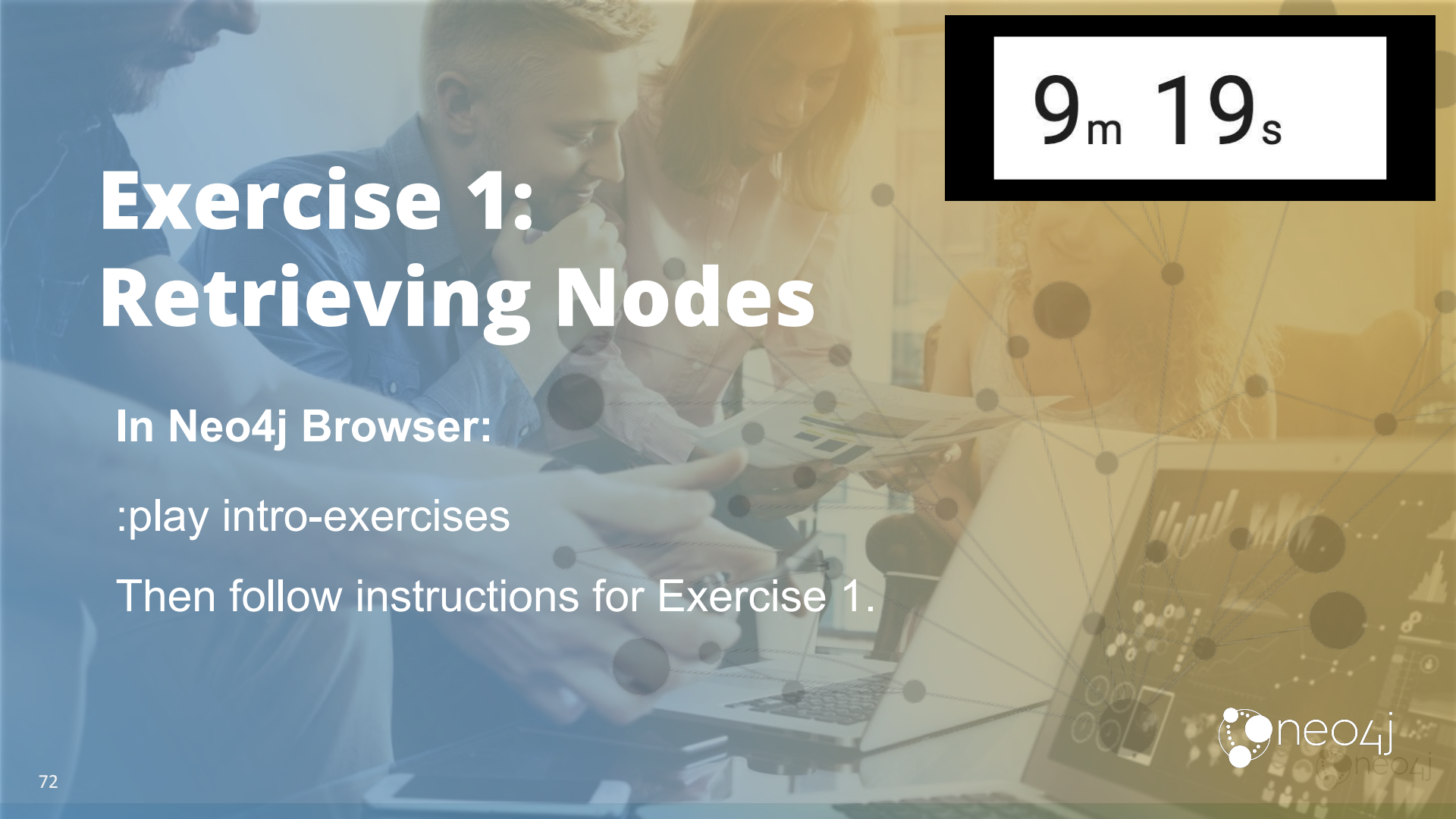
**Estimated time to complete this exercise:**

**10 minutes**

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 1.



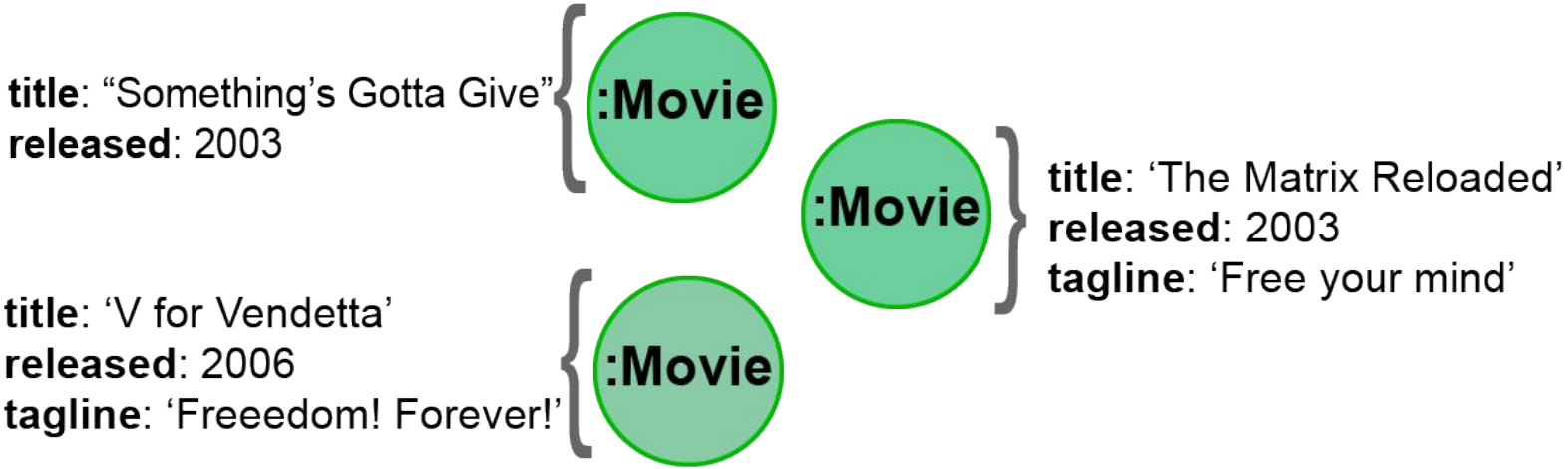

# Exercise 1: Retrieving Nodes

**In Neo4j Browser:**

:play intro-exercises

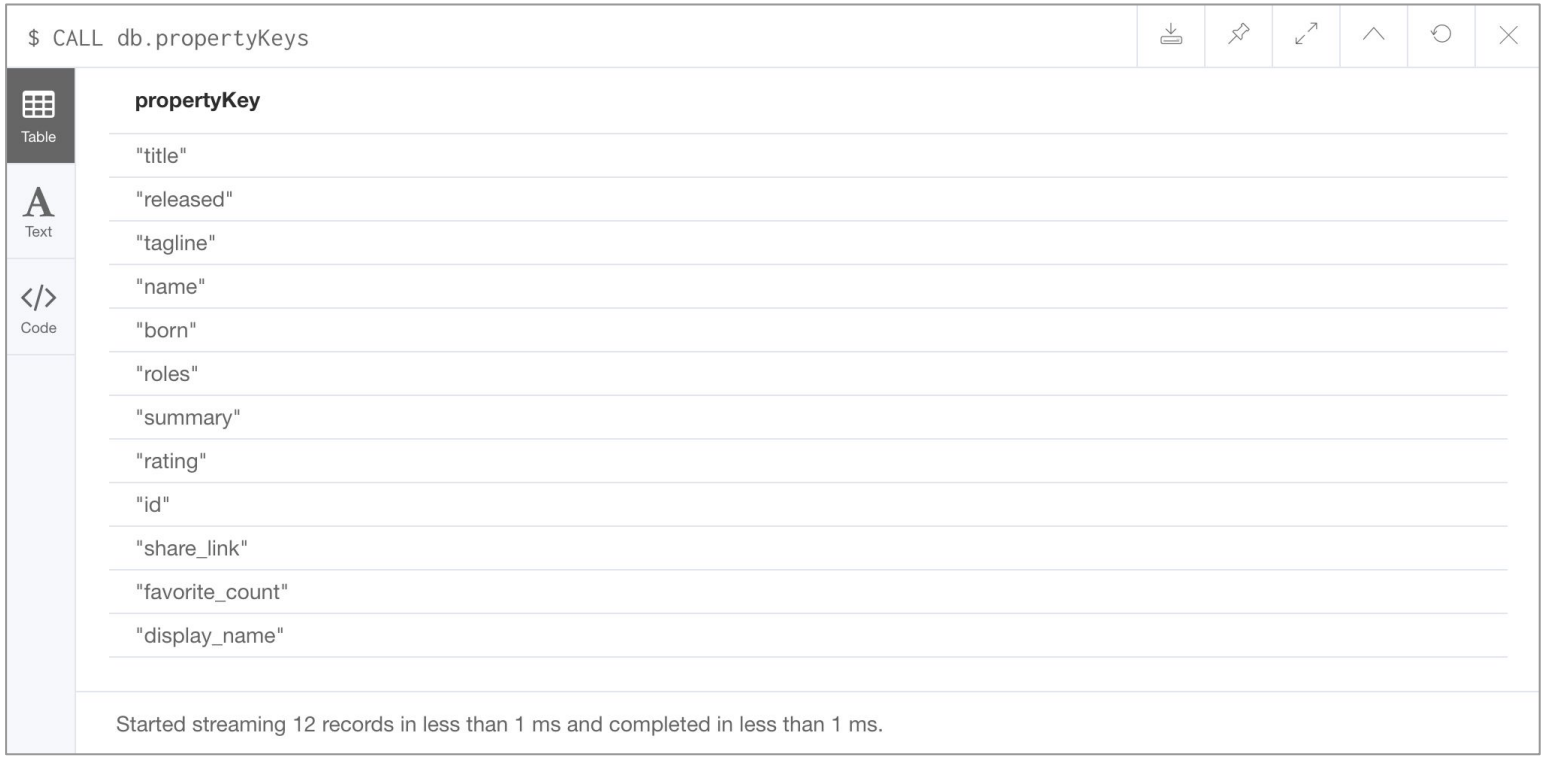
Then follow instructions for Exercise 1.

# Properties


title: “Something’s Gotta Give”
released: 2003
:Movie
:Movie
title: ‘The Matrix Reloaded’
released: 2003
tagline: ‘Free your mind’
title: ‘V for Vendetta’
released: 2006
tagline: ‘Freeedom! Forever!’
:Movie

# Examining property keys

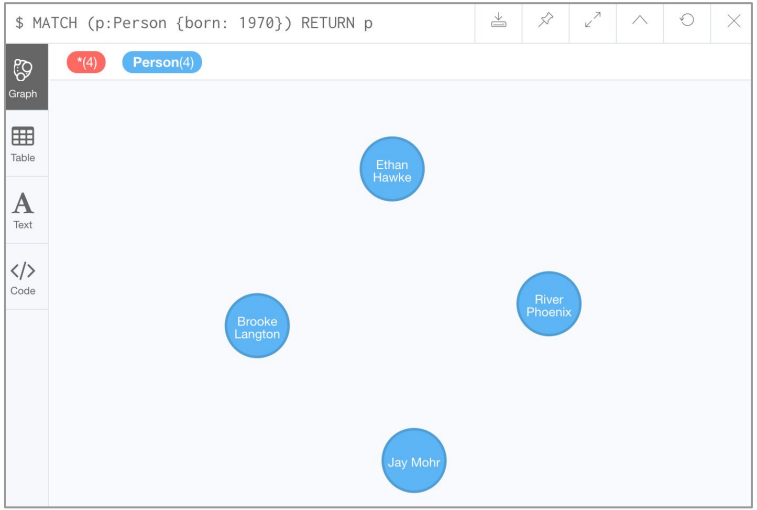
```
CALL db.propertyKeys
```



# Retrieving nodes filtered by a property value - 1

Find all *people* born in *1970, returning the nodes*:

```
MATCH (p:Person {born: 1970})
RETURN p
```



# Retrieving nodes filtered by a property value - 2

Find all movies released in *2003* with the tagline,

*Free your mind*, returning the nodes:

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

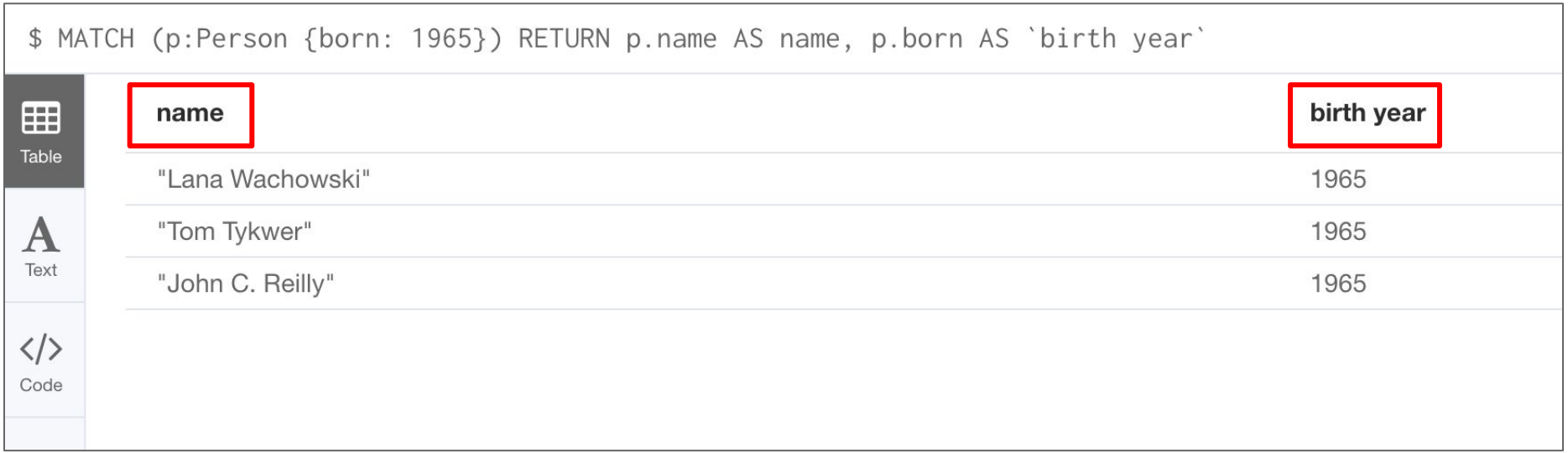
# Returning property values

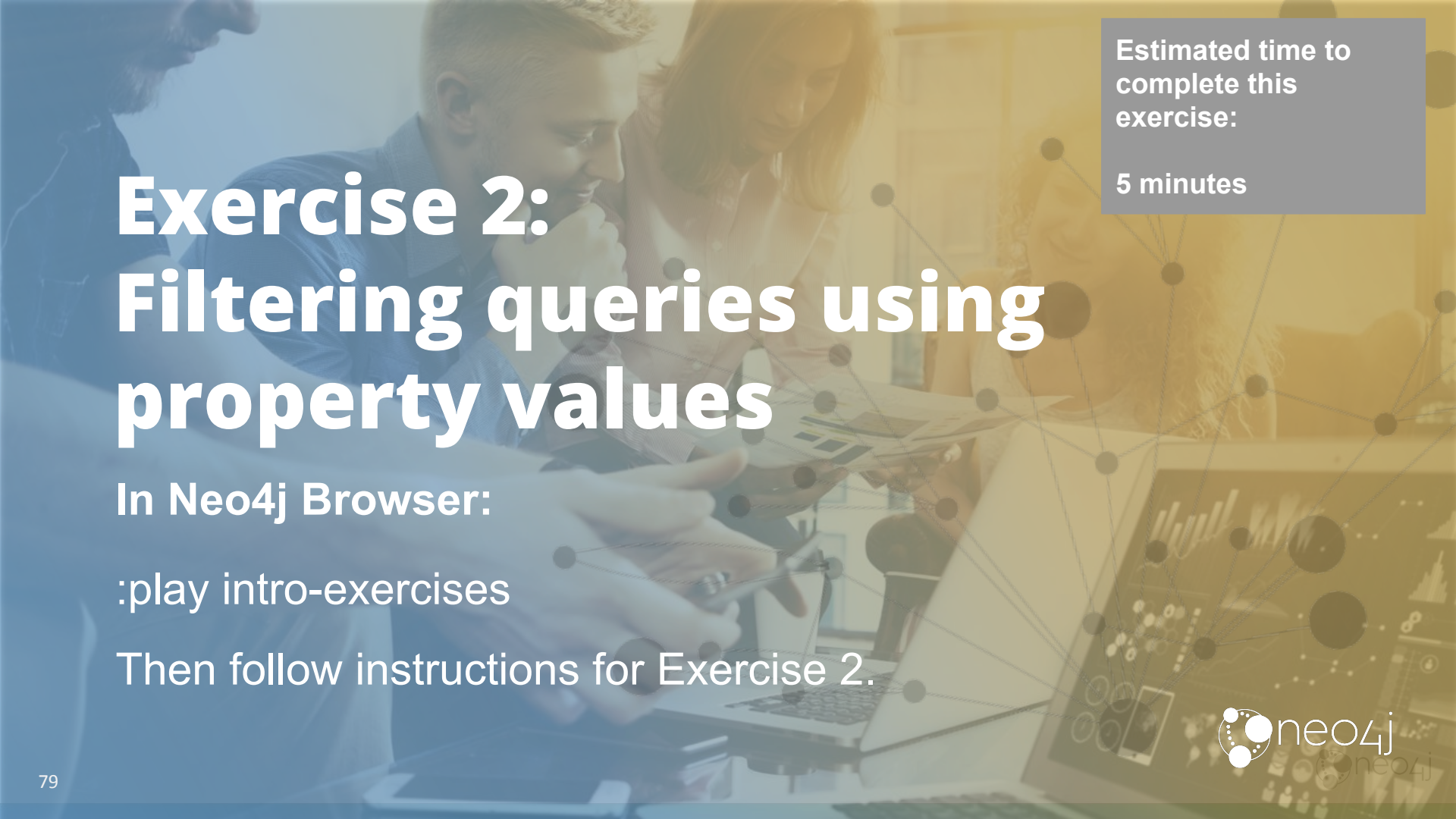
Find all people born in *1965* and return their names:

```
MATCH (p:Person {born: 1965})
RETURN p.name, p.born
```

# Specifying aliases

```
MATCH (p:Person {born: 1965})
RETURN p.name AS name, p.born AS `birth year`
```





Estimated time to complete this exercise:

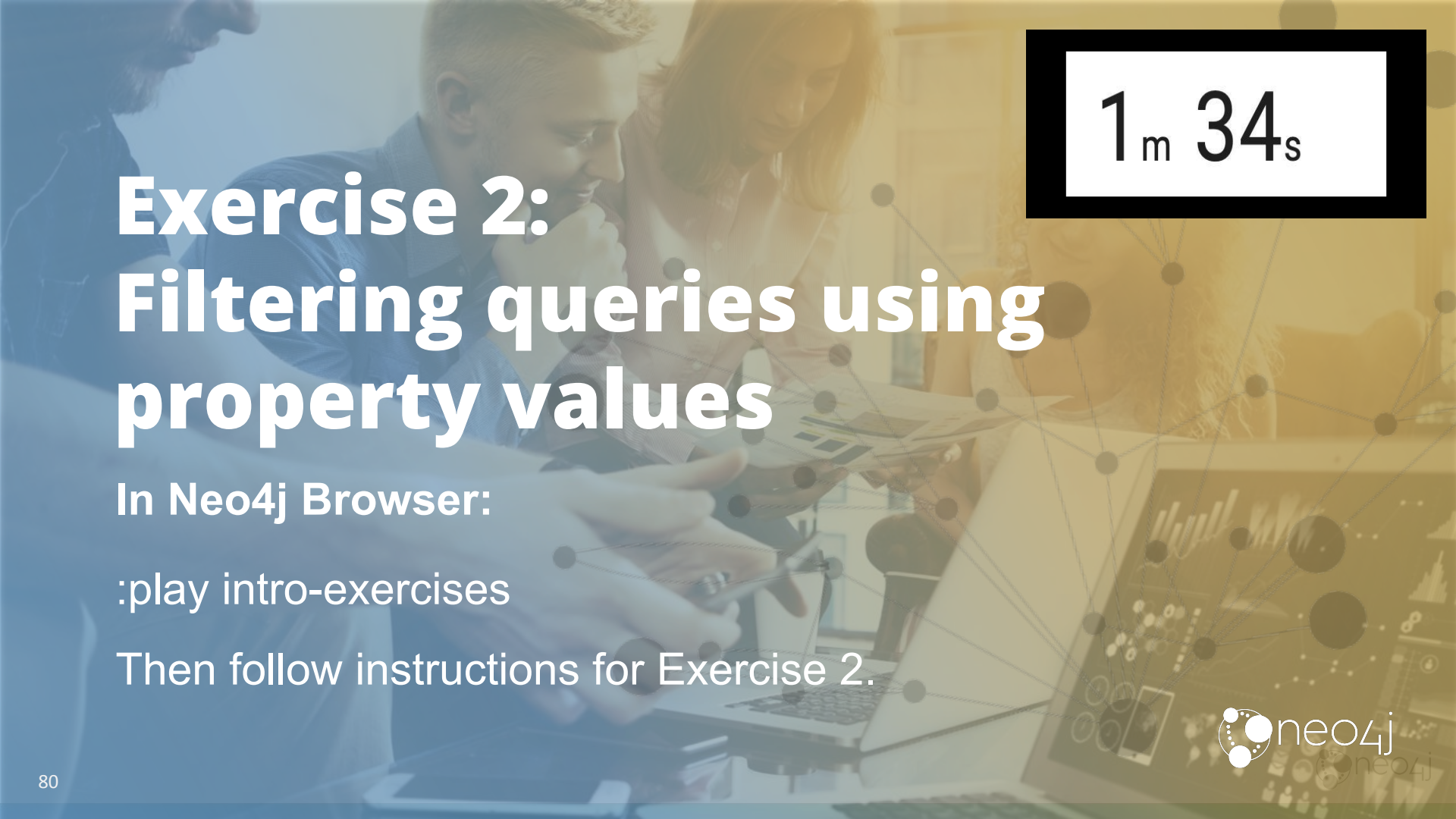
5 minutes

# Exercise 2: Filtering queries using property values

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 2.



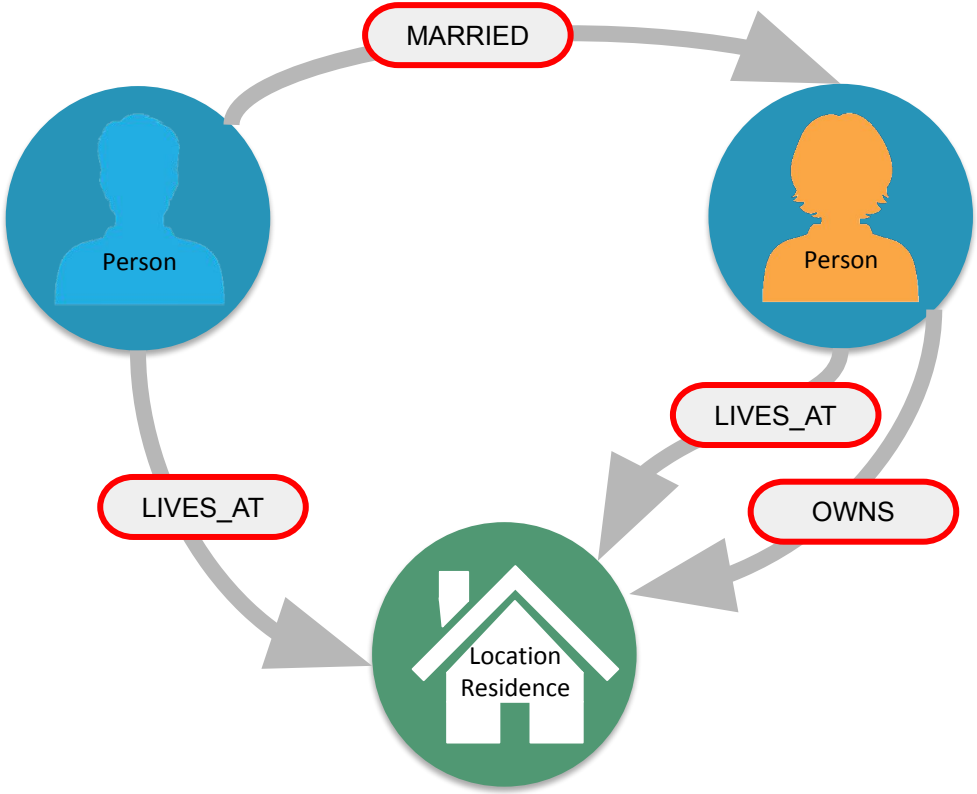

# Exercise 2: Filtering queries using property values

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 2.

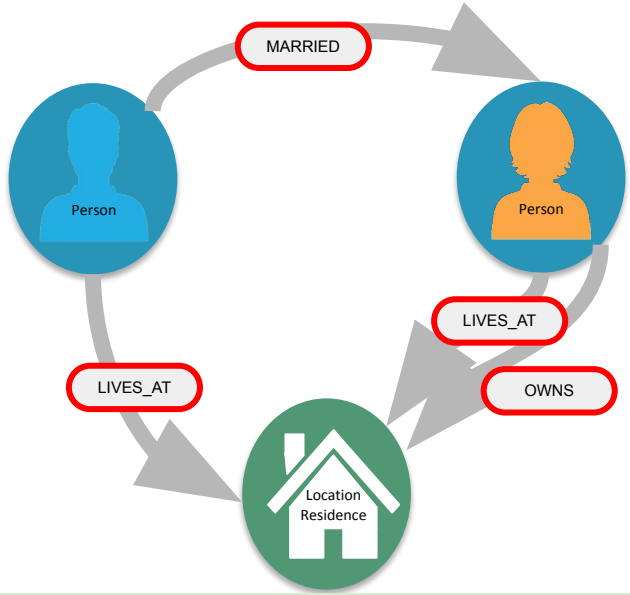
# Relationships



# ASCII art for nodes and relationships

```
()          // a node
()--()      // 2 nodes have some type of relationship
()-->()     // the first node has a relationship to the second node
()<--()     // the second node has a relationship to the first node
```

# Querying using relationships

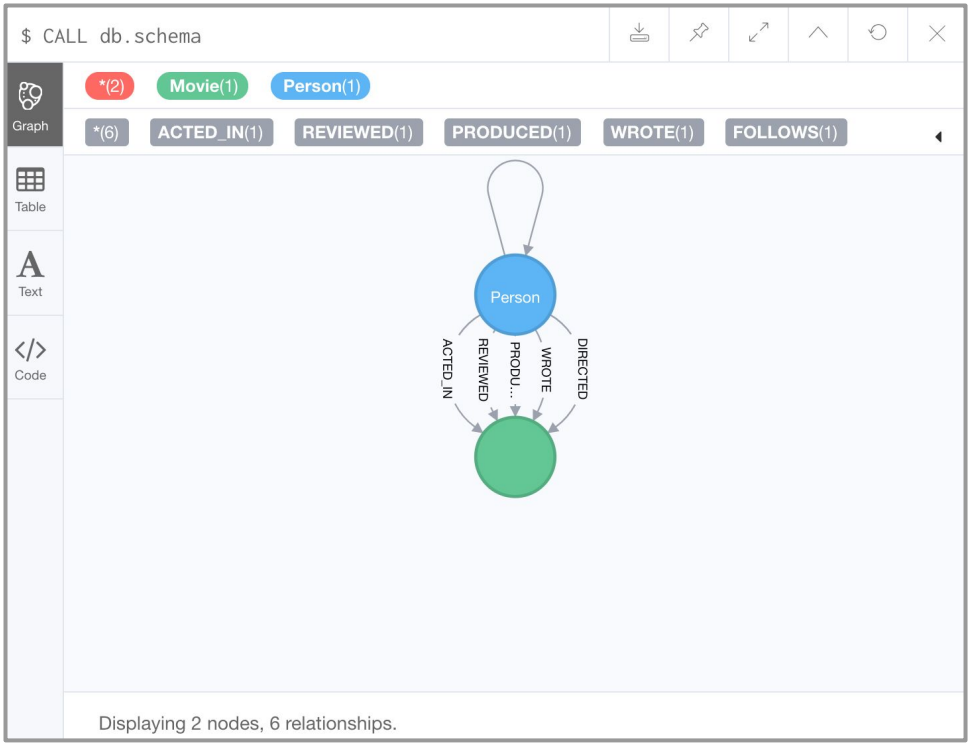



```
MATCH (p:Person)-[:LIVES_AT]->(h:Residence)
RETURN p.name, h.address
```

```
MATCH (p:Person)--(h:Residence) // any relationship
RETURN p.name, h.address
```

# Examining relationships

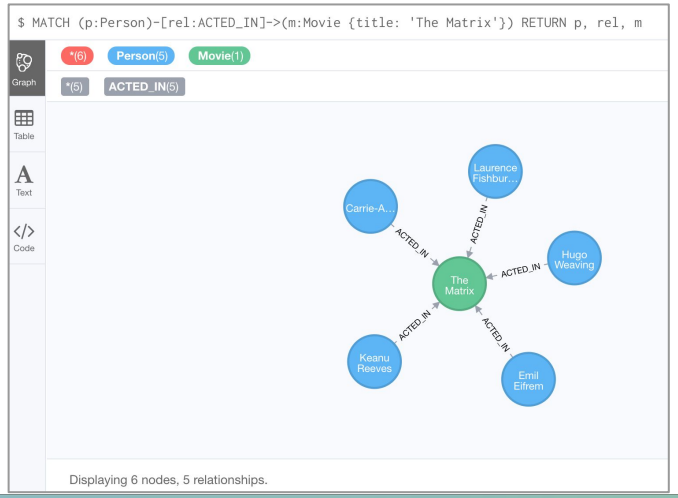
```
CALL db.schema
```



# Using a relationship in a query

Find all people who acted in the movie, *The Matrix*, returning the nodes and relationships found:

```
MATCH (p:Person)-[rel:ACTED_IN]->(m:Movie {title: 'The
Matrix'}) RETURN p, rel, m
```



# Querying by multiple relationships

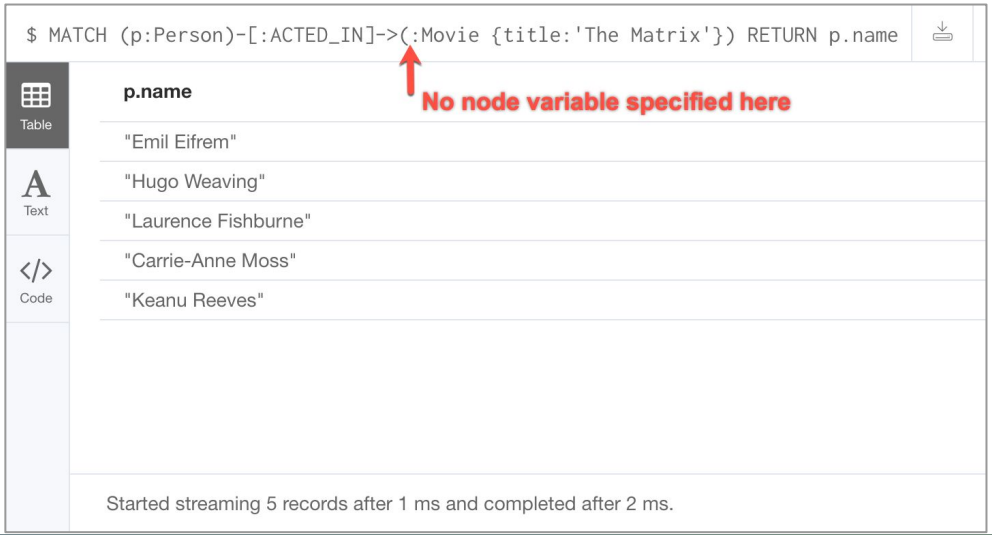
Find all movies that *Tom Hanks* acted in or directed and return the title of the move:

```
MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED_IN
|:DIRECTED]->(m:Movie)
RETURN p.name, m.title
```

# Using anonymous nodes in a query

Find all people who acted in the movie, *The Matrix* and return their names:

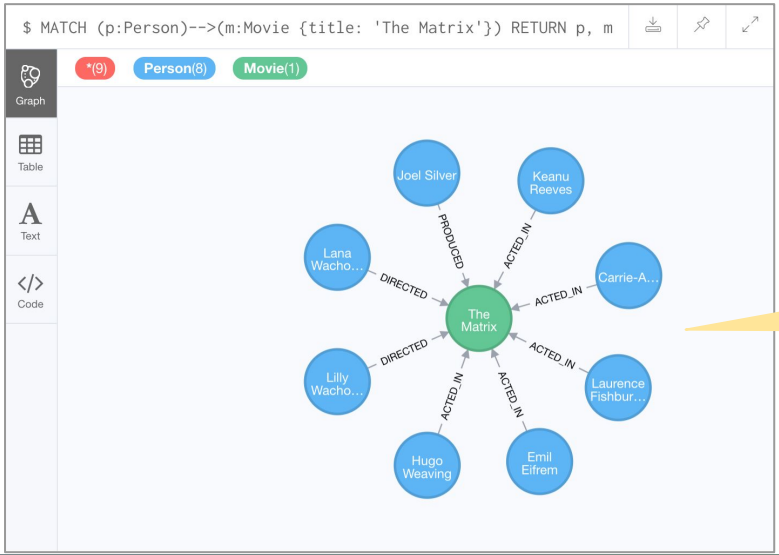
```
MATCH (p:Person)-[:ACTED_IN]->(:Movie {title: 'The Matrix'})
RETURN p.name
```



# Using an anonymous relationship for a query

Find all people who have any type of relationship to the movie, *The Matrix* and return the nodes:

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

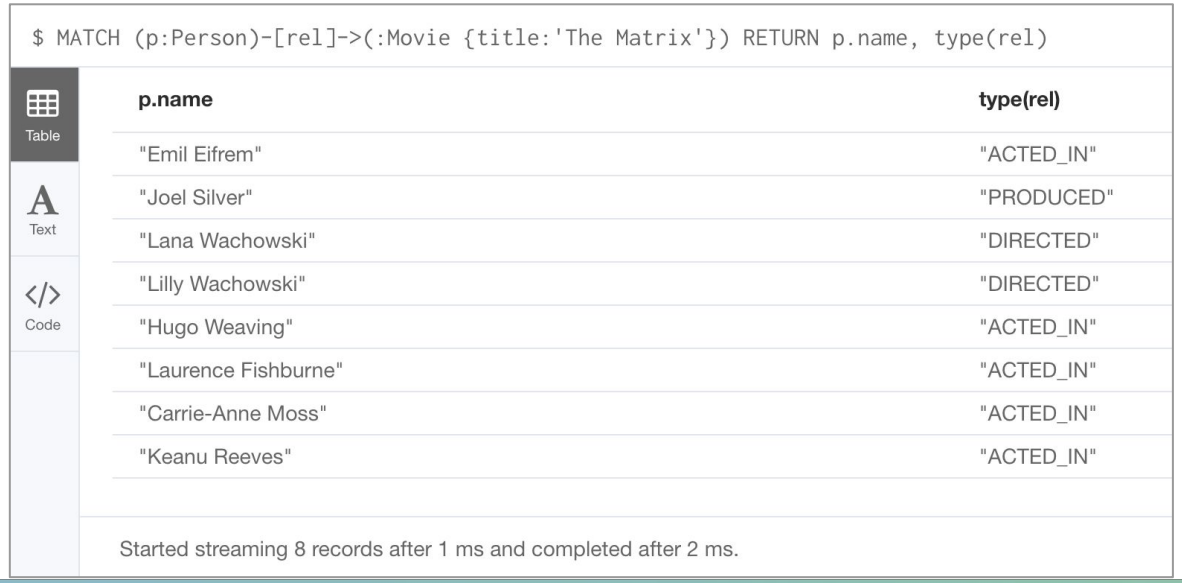



Connect result nodes enabled in Neo4j Browser

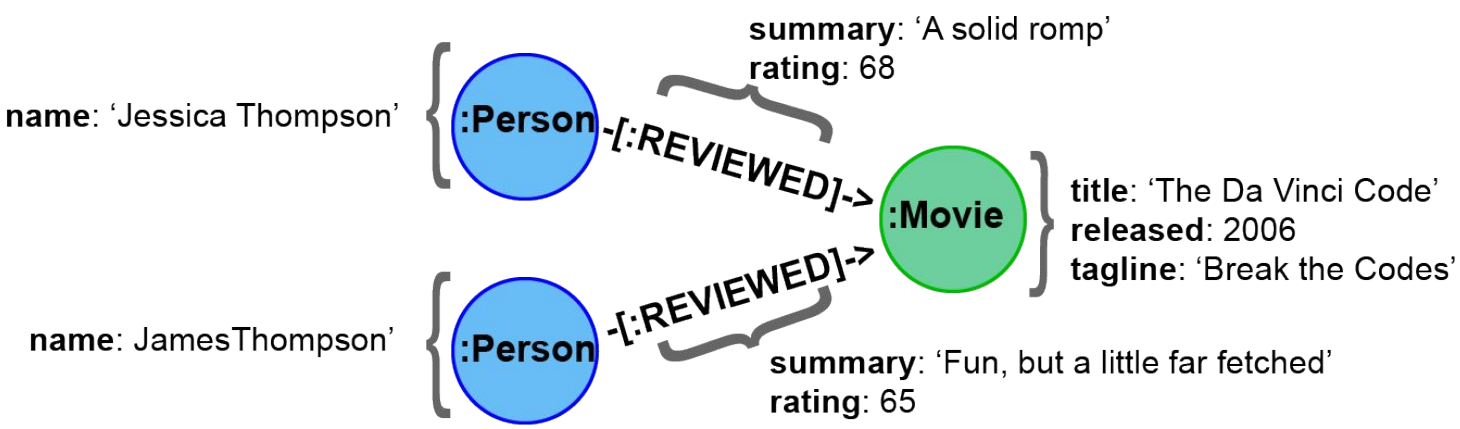
# Retrieving relationship types

Find all people who have any type of relationship to the movie, *The Matrix* and return the name of the person and their relationship type:

```
MATCH (p:Person)-[rel]->(:Movie {title:'The Matrix'})
RETURN p.name, type(rel)
```



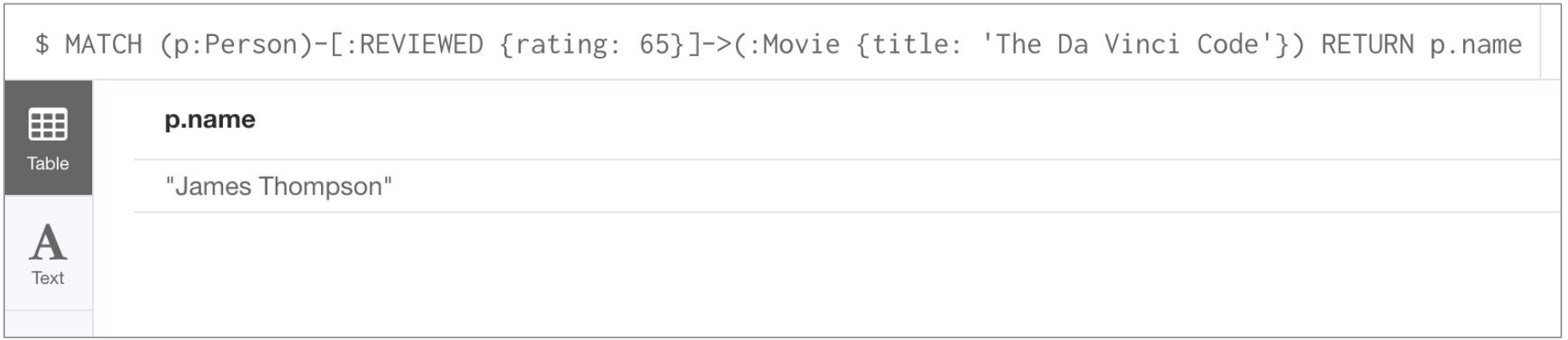
# Retrieving properties for a relationship - 1



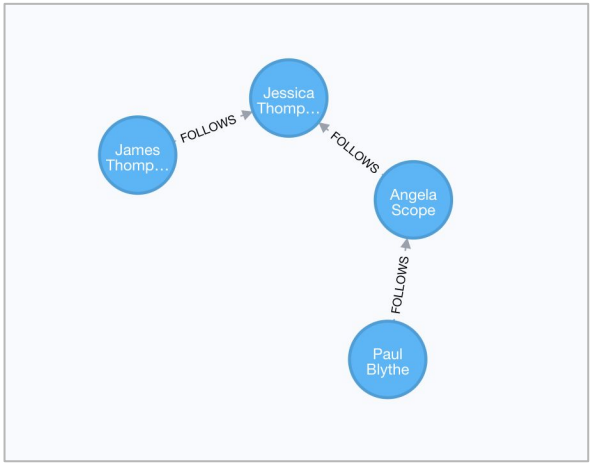
# Retrieving properties for a relationship - 2

Find all people who gave the movie, *The Da Vinci Code*, a rating of *65*, returning their names:

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

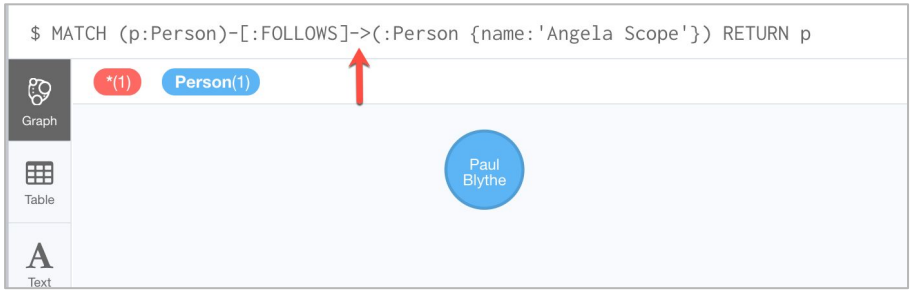


# Using patterns for queries - 1

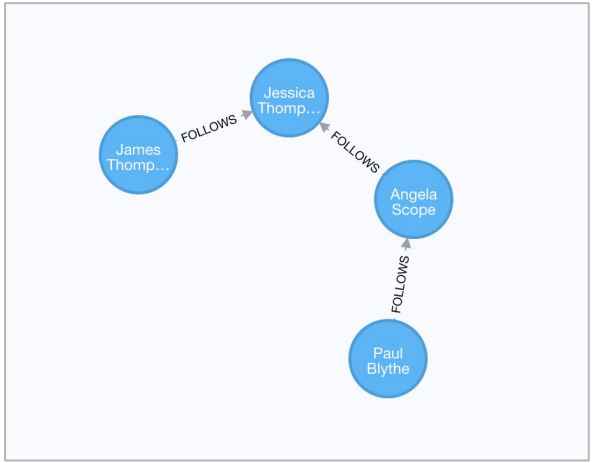



Find all people who follow *Angela Scope*, returning the nodes:

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

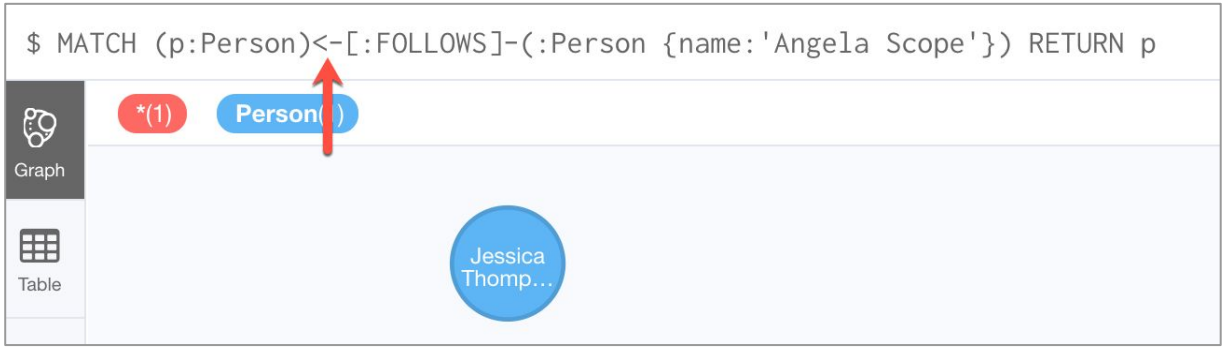


# Using patterns for queries - 2

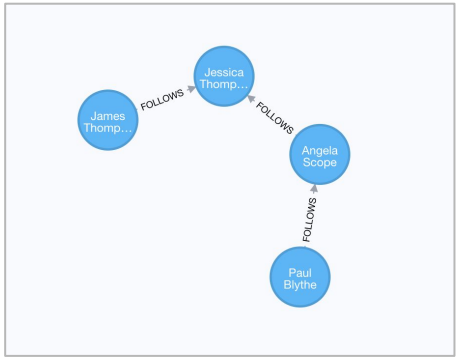



Find all people who *Angela Scope* follows, returning the nodes:

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

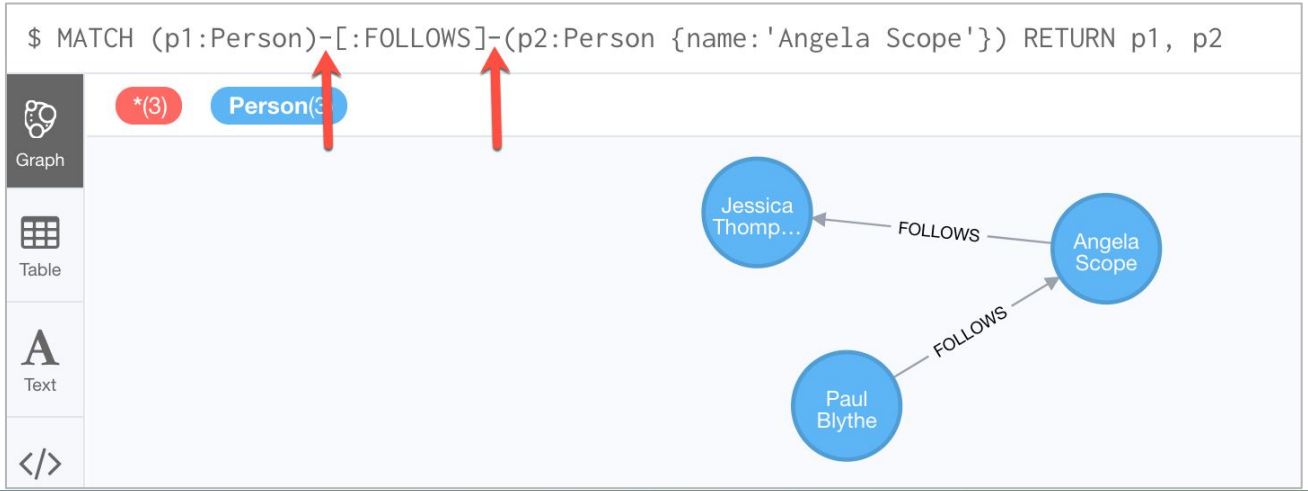


# Querying by any direction of the relationship




Find all people who follow *Angela Scope* or who *Angela Scope* follows, returning the nodes:

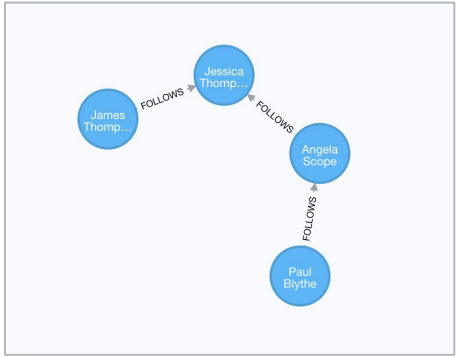
```
MATCH  (p1:Person)-[:FOLLOWS]-(p2:Person {name:'Angela Scope'})
RETURN p1, p2
```






neo4j

# Traversing relationships - 1



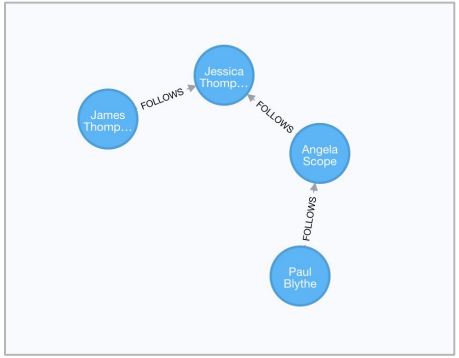

Find all people who follow anybody who follows *Jessica Thompson* returning the people as nodes:

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



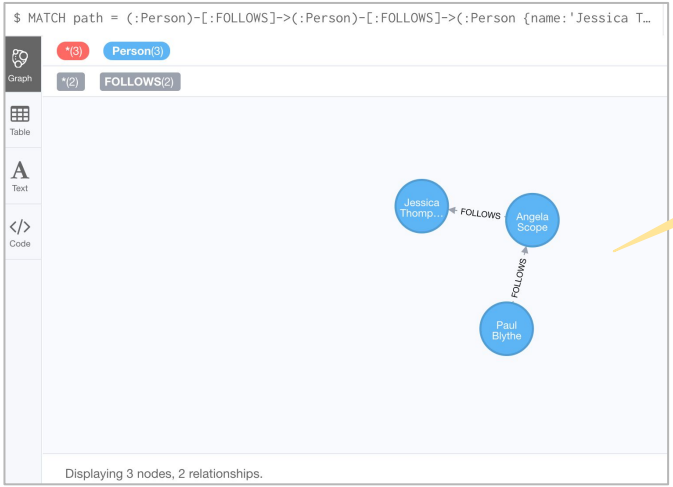

neo4j

# Traversing relationships - 2




Find the path that includes all people who follow anybody who follows *Jessica Thompson* returning the path:

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




Sub-graph

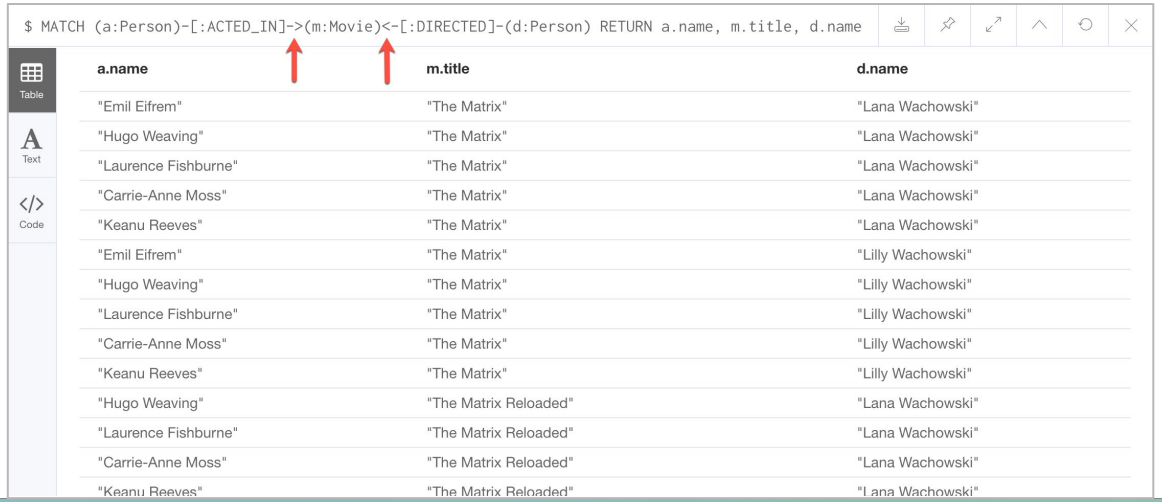


neo4j

# Using relationship direction to optimize a query

Find all people that acted in a movie and the directors for that same movie, returning the name of the actor, the movie title, and the name of the director:

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)
RETURN a.name, m.title, d.name
```




| a.name | m.title | d.name |
|---|---|---|
| "Emil Eifrem" | "The Matrix" | "Lana Wachowski" |
| "Hugo Weaving" | "The Matrix" | "Lana Wachowski" |
| "Laurence Fishburne" | "The Matrix" | "Lana Wachowski" |
| "Carrie-Anne Moss" | "The Matrix" | "Lana Wachowski" |
| "Keanu Reeves" | "The Matrix" | "Lana Wachowski" |
| "Emil Eifrem" | "The Matrix" | "Lilly Wachowski" |
| "Hugo Weaving" | "The Matrix" | "Lilly Wachowski" |
| "Laurence Fishburne" | "The Matrix" | "Lilly Wachowski" |
| "Carrie-Anne Moss" | "The Matrix" | "Lilly Wachowski" |
| "Keanu Reeves" | "The Matrix" | "Lilly Wachowski" |
| "Hugo Weaving" | "The Matrix Reloaded" | "Lana Wachowski" |
| "Laurence Fishburne" | "The Matrix Reloaded" | "Lana Wachowski" |
| "Carrie-Anne Moss" | "The Matrix Reloaded" | "Lana Wachowski" |
| "Keanu Reeves" | "The Matrix Reloaded" | "Lana Wachowski" |

# Cypher style recommendations - 1

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

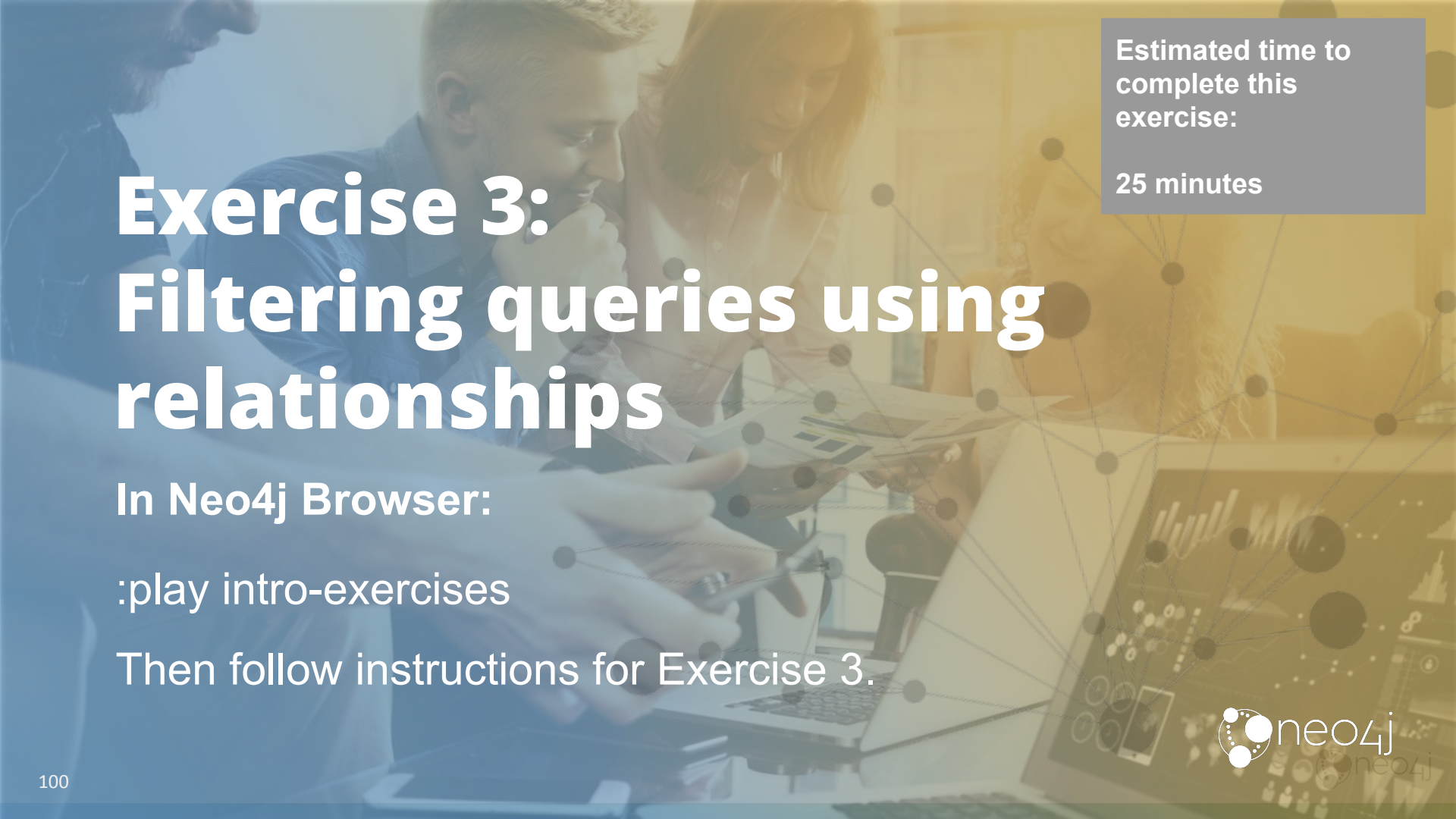
- Node labels are CamelCase and case-sensitive (examples: *Person*, *NetworkAddress*).
- Property keys, variables, parameters, aliases, and functions are camelCase case-sensitive (examples: *businessAddress*, *title*).
- Relationship types are in upper-case and can use the underscore. (examples: *ACTED_IN*, *FOLLOWS*).
- Cypher keywords are upper-case (examples: MATCH, RETURN).

# Cypher style recommendations - 2

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

- String constants are in single quotes (with exceptions).
- Specify variables only when needed for use later in the Cypher statement.
- Place named nodes and relationships (that use variables) before anonymous nodes and relationships in your MATCH clauses when possible.
- Specify anonymous relationships with -->, --, or <--

```
MATCH (:Person {name: 'Diane Keaton'})-[movRel:ACTED_IN]->
(:Movie {title:"Something's Gotta Give"})
RETURN movRel.roles
```



Estimated time to complete this exercise:

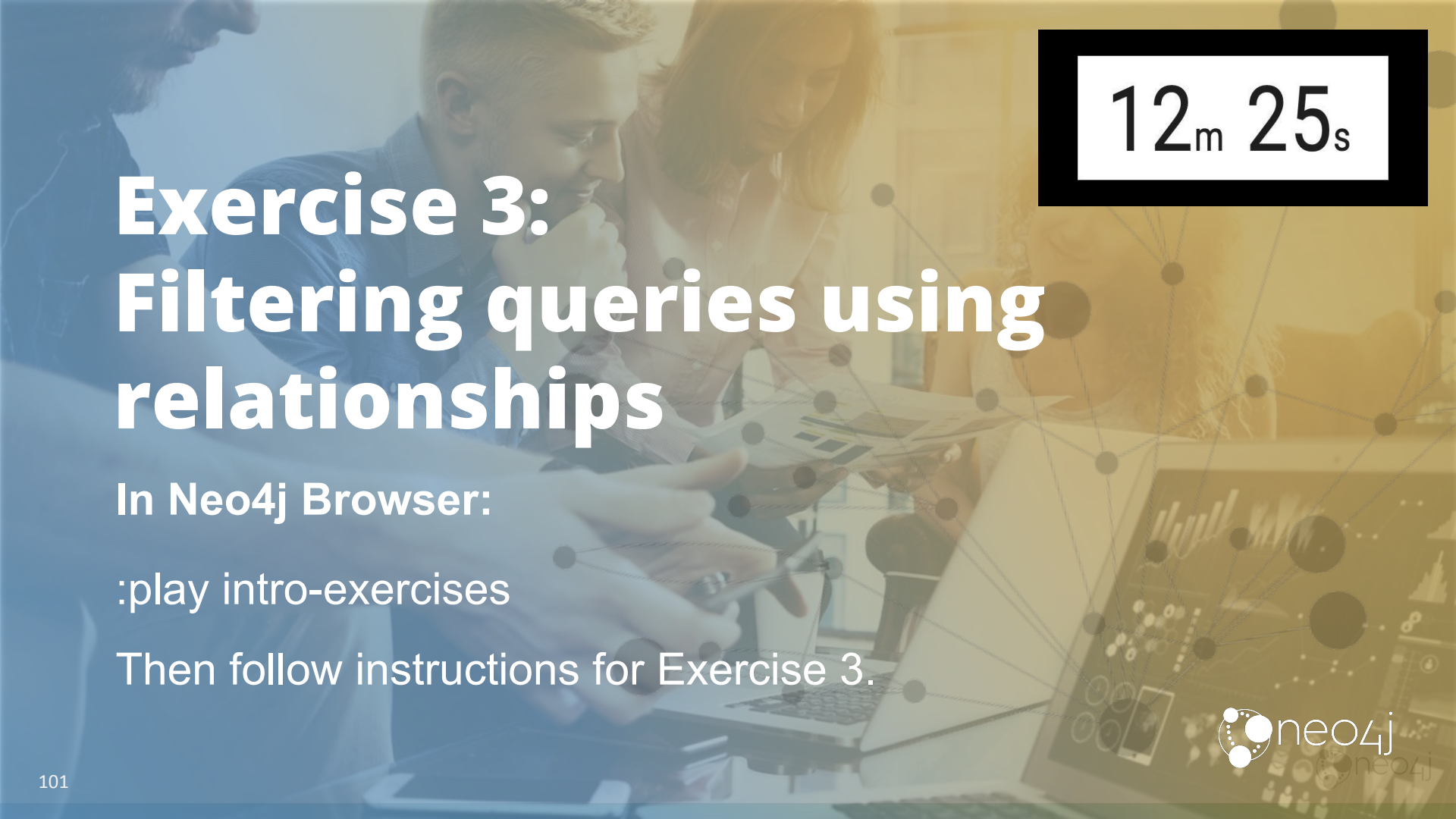
25 minutes

# Exercise 3: Filtering queries using relationships

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 3.




# Exercise 3: Filtering queries using relationships

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 3.

# Check your understanding

# Question 1

Suppose you have a graph that contains nodes representing customers and other business entities for your application. The node label in the database for a customer is *Customer*. Each *Customer* node has a property named *email* that contains the customer's email address. What Cypher query do you execute to return the email addresses for all customers in the graph?

Select the correct answer.

- ❑ `MATCH (n) RETURN n.Customer.email`
- ❑ `MATCH (c:Customer) RETURN c.email`
- ❑ `MATCH (Customer) RETURN email`
- ❑ `MATCH (c) RETURN Customer.email`

# Answer 1

Suppose you have a graph that contains nodes representing customers and other business entities for your application. The node label in the database for a customer is *Customer*. Each *Customer* node has a property named *email* that contains the customer's email address. What Cypher query do you execute to return the email addresses for all customers in the graph?

Select the correct answer.

❑ `MATCH (n) RETURN n.Customer.email`

✔ `MATCH (c:Customer) RETURN c.email`

❑ `MATCH (Customer) RETURN email`

❑ `MATCH (c) RETURN Customer.email`

# Question 2

Suppose you have a graph that contains *Customer* and *Product* nodes. A *Customer* node can have a *BOUGHT* relationship with a *Product* node. *Customer* nodes can have other relationships with *Product* nodes. A *Customer* node has a property named *customerName*. A *Product* node has a property named *productName*. What Cypher query do you execute to return all of the products (by name) bought by customer 'ABCCO'.

Select the correct answer.

❑ `MATCH (c:Customer {customerName: 'ABCCO'}) RETURN c.BOUGHT.productName`

❑ `MATCH (:Customer 'ABCCO')-[:BOUGHT]→(p:Product) RETURN p.productName`

❑ `MATCH (p:Product)←[:BOUGHT_BY]-(:Customer 'ABCCO') RETURN p.productName`

❑ `MATCH (:Customer {customerName: 'ABCCO'})-[:BOUGHT]→(p:Product) RETURN p.productName`

# Answer 2

Suppose you have a graph that contains *Customer* and *Product* nodes. A *Customer* node can have a *BOUGHT* relationship with a *Product* node. *Customer* nodes can have other relationships with *Product* nodes. A *Customer* node has a property named *customerName*. A *Product* node has a property named *productName*. What Cypher query do you execute to return all of the products (by name) bought by customer 'ABCCO'.

Select the correct answer.

❏ `MATCH (c:Customer {customerName: 'ABCCO'}) RETURN c.BOUGHT.productName`

❏ `MATCH (:Customer 'ABCCO')-[:BOUGHT]→(p:Product) RETURN p.productName`

❏ `MATCH (p:Product)←[:BOUGHT_BY]-(:Customer 'ABCCO') RETURN p.productName`

✔ `MATCH (:Customer {customerName: 'ABCCO'})-[:BOUGHT]→(p:Product) RETURN p.productName`

# Question 3

When must you use a variable in a MATCH clause?

Select the correct answer.

- ❏ When you want to query the graph using a node label.
- ❏ When you specify a property value to match the query.
- ❏ When you want to use the node or relationship to return a result.
- ❏ When the query involves two types of nodes.

# Answer 3

When must you use a variable in a MATCH clause?

Select the correct answer.

- ❏ When you want to query the graph using a node label.
- ❏ When you specify a property value to match the query.
- ✔ When you want to use the node or relationship to return a result.
- ❏ When the query involves two types of nodes.

# Summary

You should be able to write Cypher statements to:

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

# Getting More Out of Queries

# Overview

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

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

# Filtering queries using WHERE

Previously you retrieved nodes as follows:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie {released: 2008})
RETURN p, m
```

A more flexible syntax for the same query is:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.released = 2008
RETURN p, m
```

Testing more than equality:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.released = 2008 OR m.released = 2009
RETURN p, m
```

# Specifying ranges in WHERE clauses

This query to find all people who acted in movies released between 2003 and 2004:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.released >= 2003 AND m.released <= 2004
RETURN p.name, m.title, m.released
```

Is the same as:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE 2003 <= m.released <= 2004
RETURN p.name, m.title, m.released
```

# Testing labels

These queries:

```
MATCH (p:Person)
RETURN p.name
```

```
MATCH (p:Person)-[:ACTED_IN]->(:Movie {title: 'The
Matrix'})
RETURN p.name
```

Can be rewritten as:

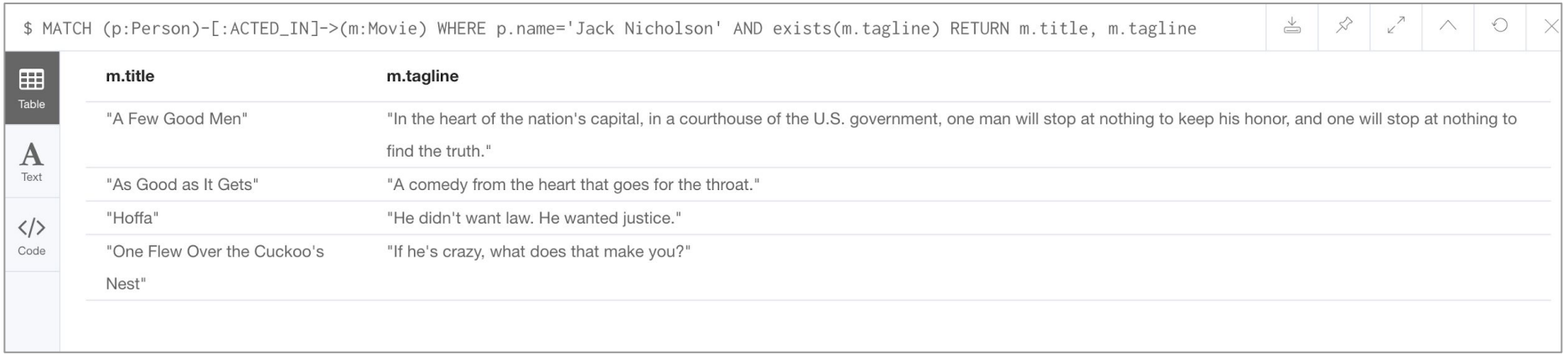
```
MATCH (p)
WHERE p:Person
RETURN p.name
```

```
MATCH (p)-[:ACTED_IN]->(m)
WHERE p:Person AND m:Movie AND m.title='The Matrix'
RETURN p.name
```

# Testing the existence of a property

Find all movies that *Jack Nicholson* acted in that have a tagline, returning the title and tagline of the movie:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name='Jack Nicholson' AND exists(m.tagline)
RETURN m.title, m.tagline
```

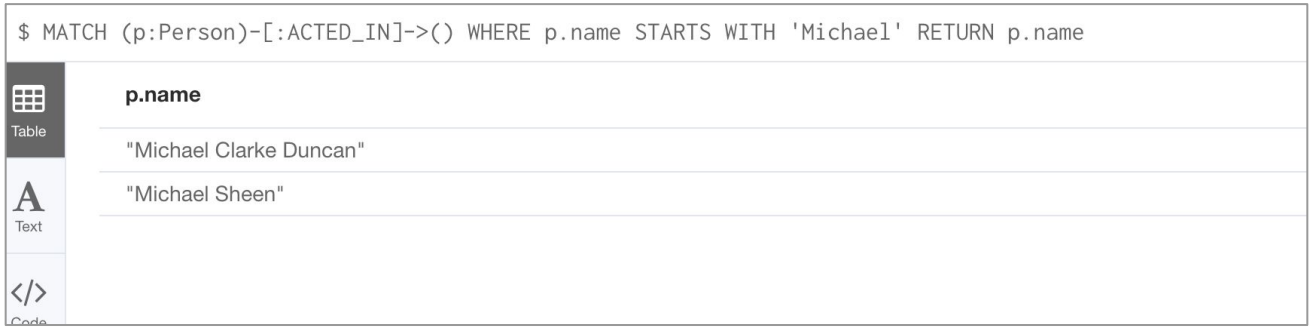



| m.title | m.tagline |
|---|---|
| "A Few Good Men" | "In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth." |
| "As Good as It Gets" | "A comedy from the heart that goes for the throat." |
| "Hoffa" | "He didn't want law. He wanted justice." |
| "One Flew Over the Cuckoo's Nest" | "If he's crazy, what does that make you?" |

# Testing strings

Find all actors whose name begins with *Michael*:

```
MATCH (p:Person)-[:ACTED_IN]->()
WHERE p.name STARTS WITH 'Michael'
RETURN p.name
```

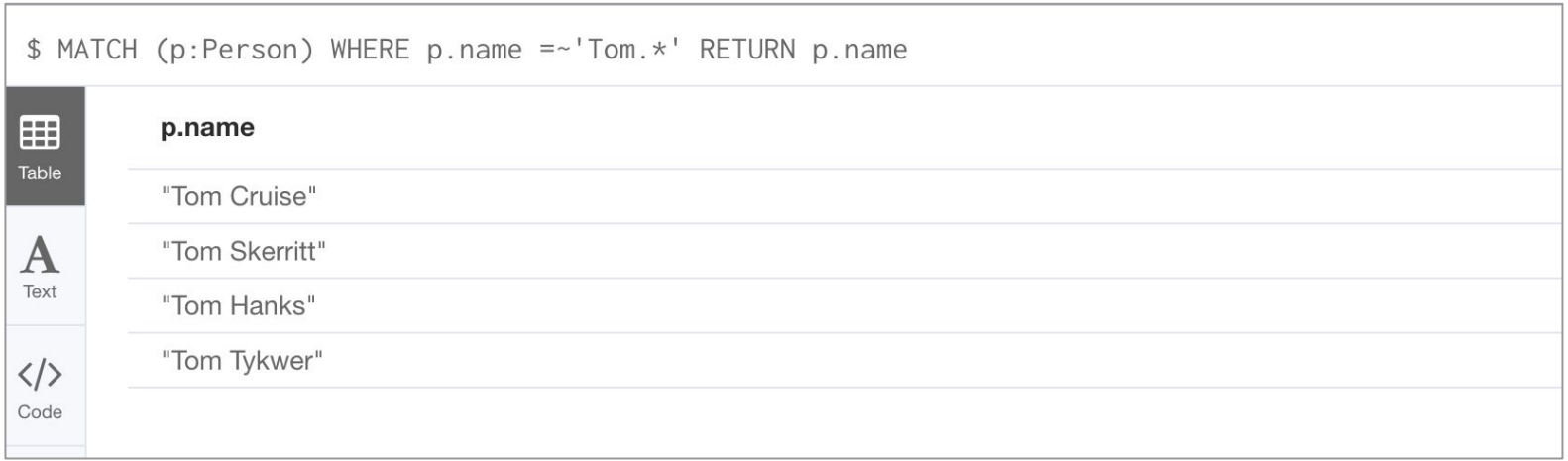



```
MATCH (p:Person)-[:ACTED_IN]->()
WHERE toLower(p.name) STARTS WITH 'michael'
RETURN p.name
```

# Testing with regular expressions

Find  people whose name starts with *Tom*:

```
MATCH (p:Person)
WHERE p.name =~'Tom.*'
RETURN p.name
```



# Testing with patterns - 1

Find all people who wrote movies returning their names and the title of the movie they wrote:

```
MATCH (p:Person)-[:WROTE]->(m:Movie)
RETURN p.name, m.title
```

$ MATCH (p:Person)-[:WROTE]->(m:Movie) RETURN p.name, m.title

Table | Text | Code

| p.name | m.title |
|---|---|
| "Aaron Sorkin" | "A Few Good Men" |
| "Jim Cash" | "Top Gun" |
| "Cameron Crowe" | "Jerry Maguire" |
| "Nora Ephron" | "When Harry Met Sally" |
| "David Mitchell" | "Cloud Atlas" |
| "Lilly Wachowski" | "V for Vendetta" |
| "Lana Wachowski" | "V for Vendetta" |
| "Lana Wachowski" | "Speed Racer" |
| "Lilly Wachowski" | "Speed Racer" |
| "Nancy Meyers" | "Something's Gotta Give" |

Started streaming 10 records in less than 1 ms and completed after 1 ms.

# Testing with patterns - 2

Find the people who wrote movies, but did not direct them, returning their names and the title of the movie:

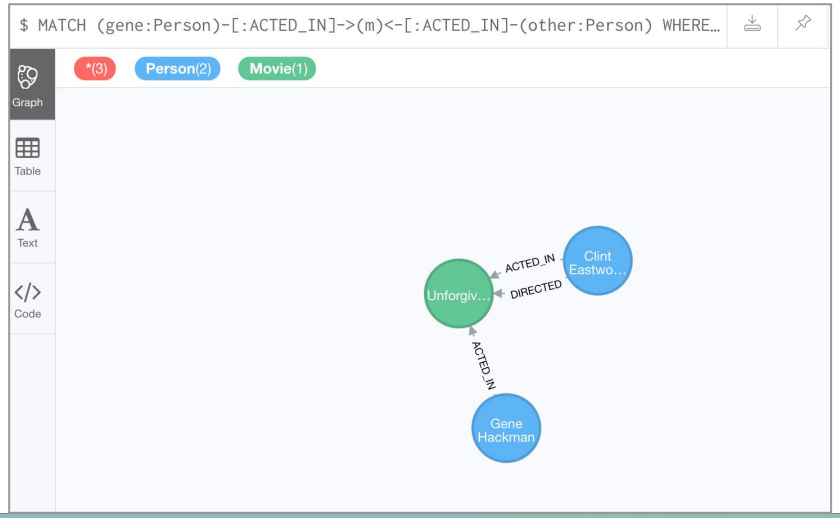
```
MATCH (p:Person)-[:WROTE]->(m:Movie)
WHERE NOT exists( (p)-[:DIRECTED]->(m) )
RETURN p.name, m.title
```


| p.name | m.title |
|---|---|
| "Aaron Sorkin" | "A Few Good Men" |
| "Jim Cash" | "Top Gun" |
| "Nora Ephron" | "When Harry Met Sally" |
| "David Mitchell" | "Cloud Atlas" |
| "Lana Wachowski" | "V for Vendetta" |
| "Lilly Wachowski" | "V for Vendetta" |

# Testing with patterns - 3

Find *Gene Hackman* and the movies that he acted in with another person who also directed the movie, returning the nodes found:

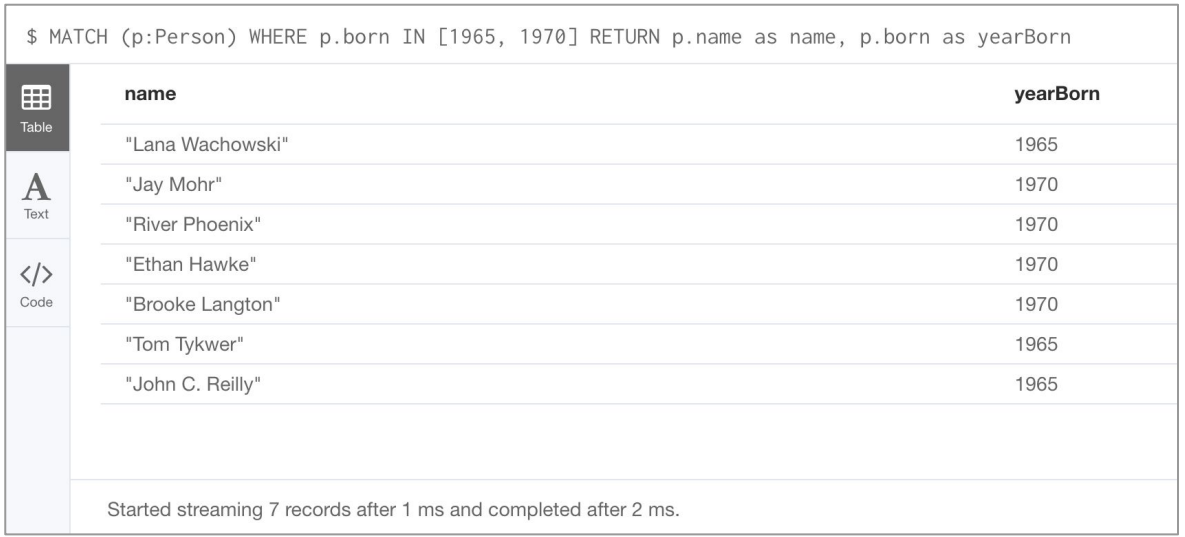
```
MATCH (gene:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(other:Person)
WHERE gene.name= 'Gene Hackman' AND exists( (other)-[:DIRECTED]->(m) )
RETURN  gene, other, m
```



# Testing with list values - 1

Find all people born in 1965 and 1970:

```
MATCH (p:Person)
WHERE p.born IN [1965, 1970]
RETURN p.name as name, p.born as yearBorn
```



# Testing with list values - 2

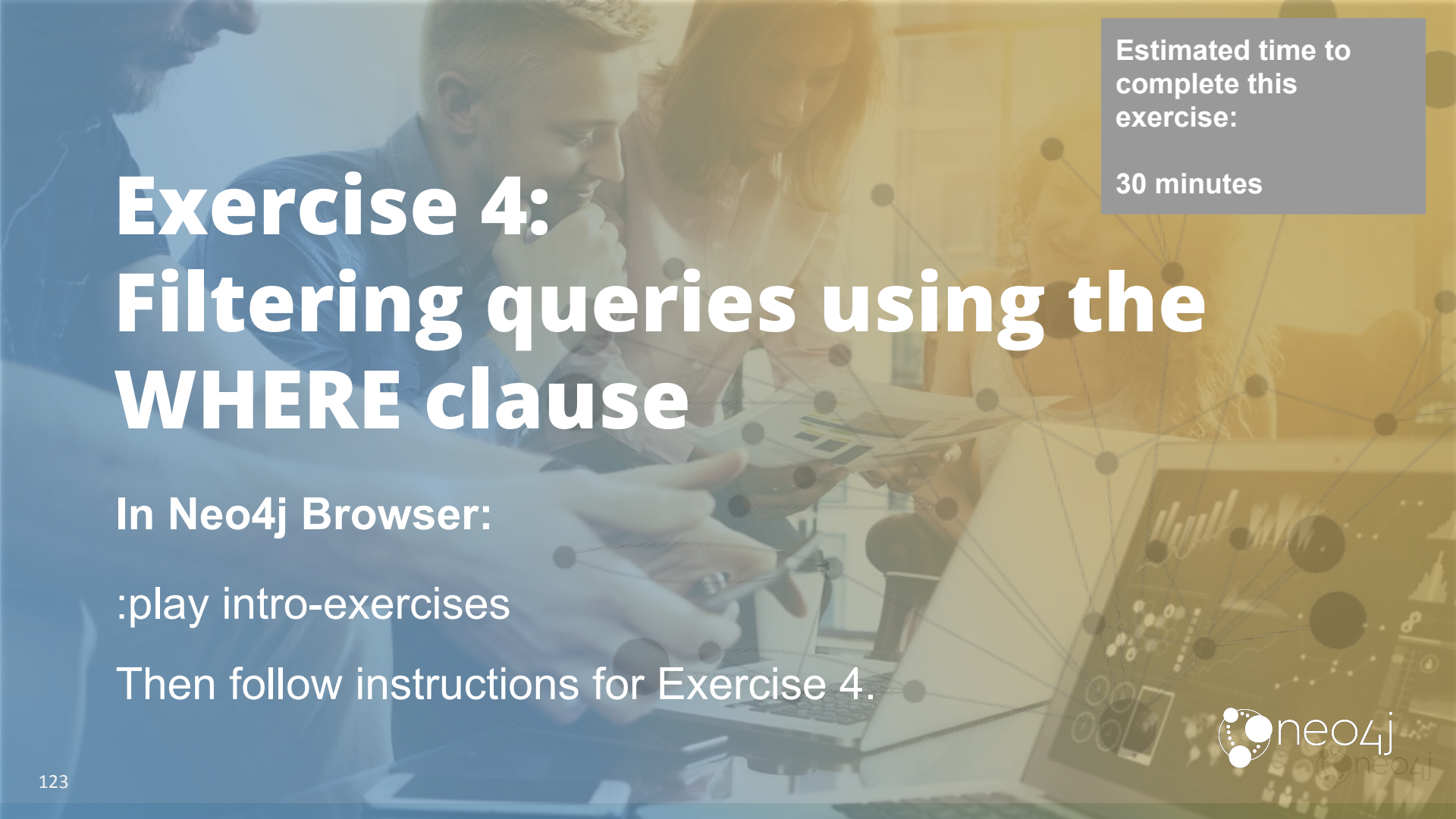
Find the actor who played *Neo* in the movie, *The Matrix*:

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE  'Neo' IN r.roles AND m.title='The Matrix'
RETURN p.name
```

```
$ MATCH (p:Person)-[r:ACTED_IN]->(m:Movie) WHERE "Neo" IN r.roles and m.title="The Matrix" RETURN p.name
```

Table

| p.name |
| --- |
| "Keanu Reeves" |



Estimated time to complete this exercise:

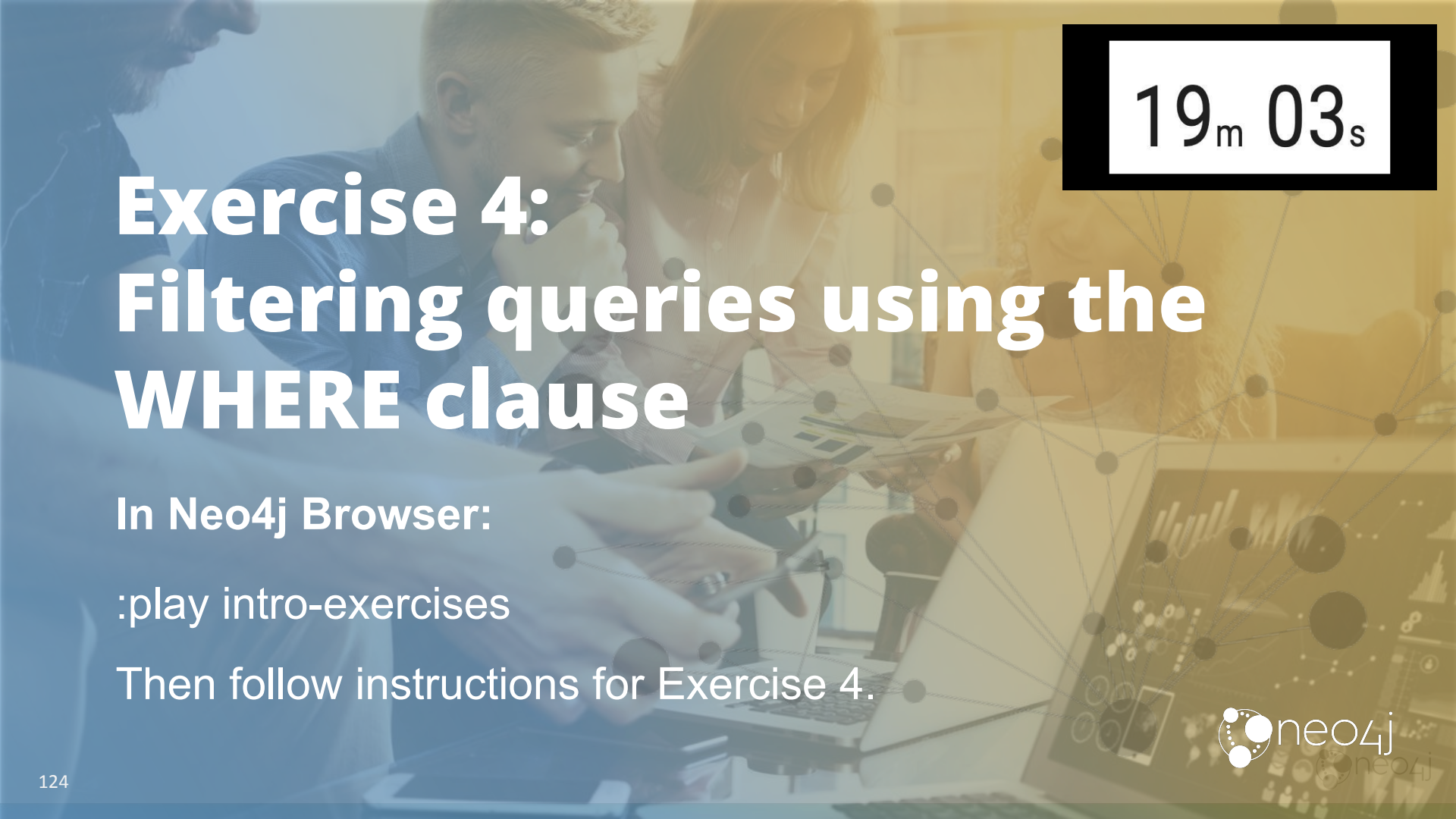
30 minutes

# Exercise 4: Filtering queries using the WHERE clause

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 4.




# Exercise 4: Filtering queries using the WHERE clause

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 4.

# Controlling query processing

- Multiple MATCH clauses
- Varying length paths
- Collecting results into lists
- Counting results

# Specifying multiple MATCH patterns

This query to find people who either acted or directed a movie released in *2000* is specified with two MATCH patterns:

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie),
      (m:Movie)<-[:DIRECTED]-(d:Person)
WHERE m.released = 2000
RETURN a.name, m.title, d.name
```

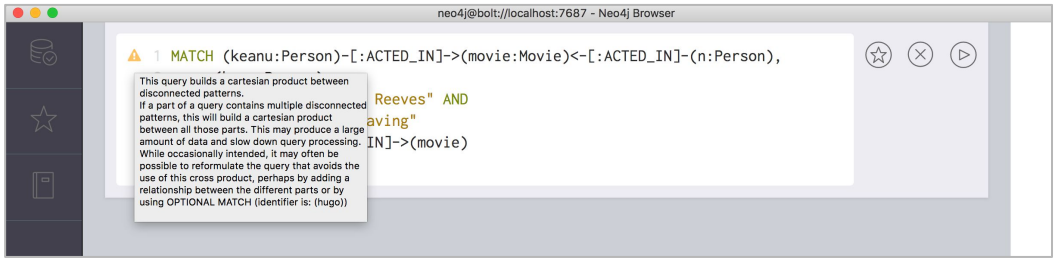
A best practice is to use a single MATCH pattern if possible:

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)
WHERE m.released = 2000
RETURN a.name, m.title, d.name
```

# Example 1: Using two MATCH patterns

Find the actors who acted in the same movies as *Keanu Reeves*, but not when *Hugo Weaving* acted in the same movie:

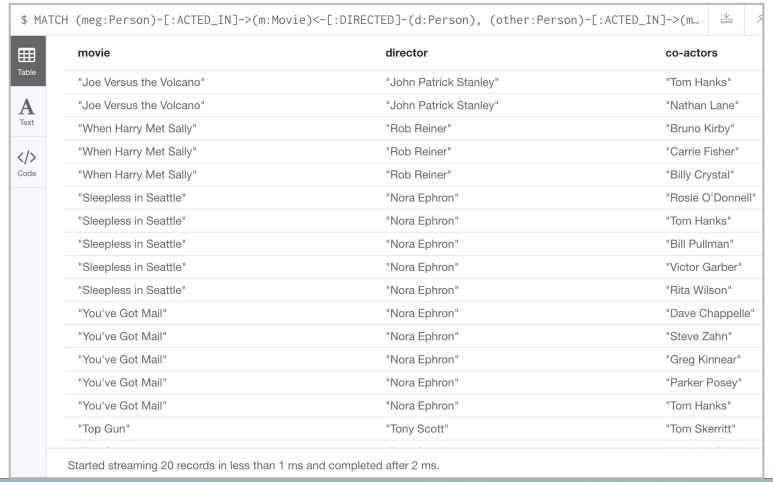
```
MATCH (keanu:Person)-[:ACTED_IN]->(movie:Movie)<-[:ACTED_IN]-(n:Person), (hugo:Person)
WHERE keanu.name='Keanu Reeves' AND hugo.name='Hugo Weaving' AND
      NOT (hugo)-[:ACTED_IN]->(movie)
RETURN n.name
```




# Example 2: Using two MATCH patterns

Retrieve the movies that *Meg Ryan* acted in and their respective directors, as well as the other actors that acted in these movies:

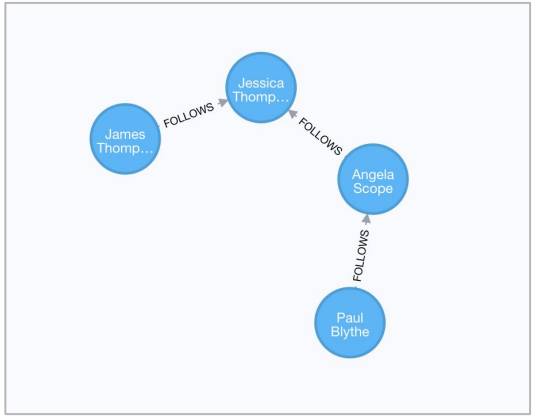
```
MATCH (meg:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person),
      (other:Person)-[:ACTED_IN]->(m)
WHERE meg.name = 'Meg Ryan'
RETURN m.title as movie, d.name AS director , other.name AS `co-actors`
```




| movie | director | co-actors |
|---|---|---|
| "Joe Versus the Volcano" | "John Patrick Stanley" | "Tom Hanks" |
| "Joe Versus the Volcano" | "John Patrick Stanley" | "Nathan Lane" |
| "When Harry Met Sally" | "Rob Reiner" | "Bruno Kirby" |
| "When Harry Met Sally" | "Rob Reiner" | "Carrie Fisher" |
| "When Harry Met Sally" | "Rob Reiner" | "Billy Crystal" |
| "Sleepless in Seattle" | "Nora Ephron" | "Rosie O'Donnell" |
| "Sleepless in Seattle" | "Nora Ephron" | "Tom Hanks" |
| "Sleepless in Seattle" | "Nora Ephron" | "Bill Pullman" |
| "Sleepless in Seattle" | "Nora Ephron" | "Victor Garber" |
| "Sleepless in Seattle" | "Nora Ephron" | "Rita Wilson" |
| "You've Got Mail" | "Nora Ephron" | "Dave Chappelle" |
| "You've Got Mail" | "Nora Ephron" | "Steve Zahn" |
| "You've Got Mail" | "Nora Ephron" | "Greg Kinnear" |
| "You've Got Mail" | "Nora Ephron" | "Parker Posey" |
| "You've Got Mail" | "Nora Ephron" | "Tom Hanks" |
| "Top Gun" | "Tony Scott" | "Tom Skerritt" |

Started streaming 20 records in less than 1 ms and completed after 2 ms.

# Specifying varying length paths



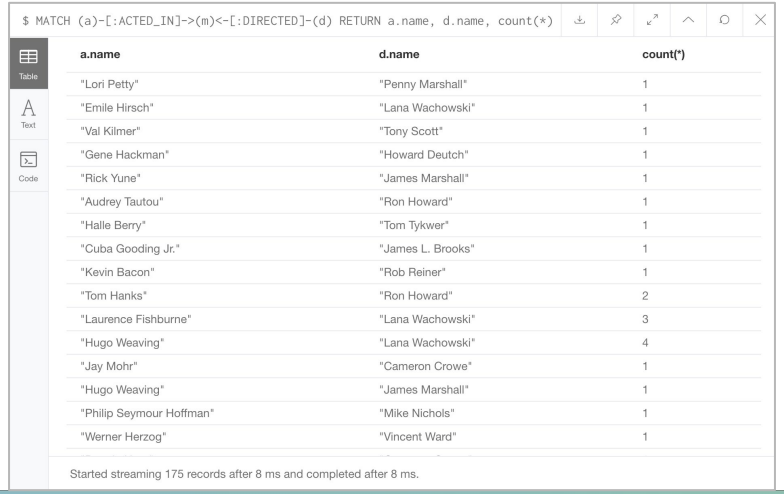

Find all people who are exactly two hops away from *Paul Blythe*:

```
MATCH (follower:Person)-[:FOLLOWS*2]->(p:Person)
WHERE follower.name = 'Paul Blythe'
RETURN p
```

# Aggregation in Cypher

- Different from SQL - no need to specify a grouping key.
- As soon as you use an aggregation function, all non-aggregated result columns automatically become grouping keys.
- Implicit grouping based upon fields in the RETURN clause.

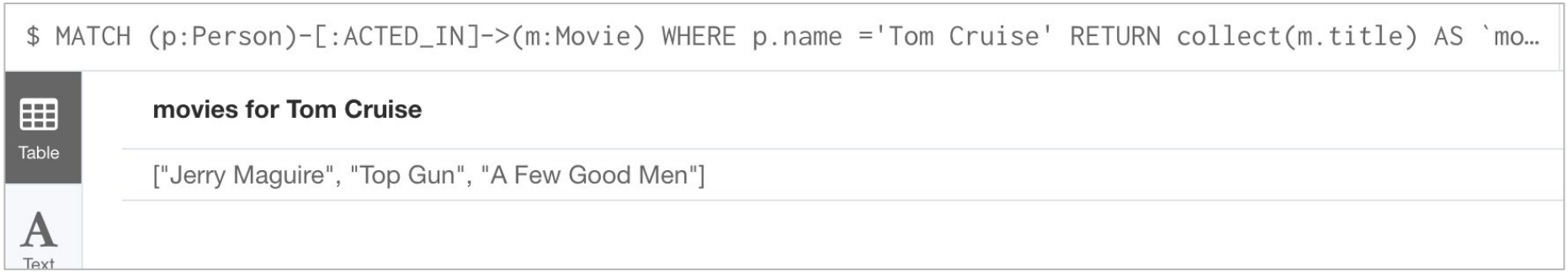
```
// implicitly groups by a.name and d.name
MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
RETURN a.name, d.name, count(*)
```



# Collecting results

Find the movies that Tom Cruise acted in and return them as a list:

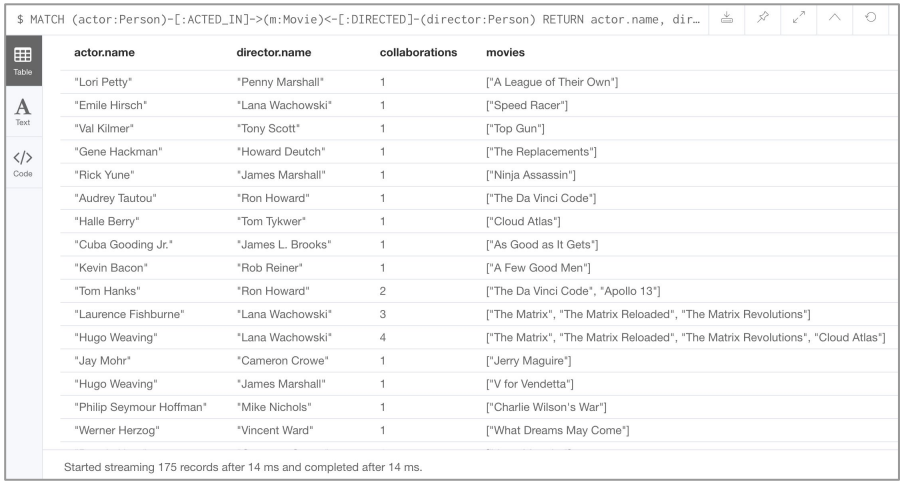
```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name ='Tom Cruise'
RETURN collect(m.title) AS `movies for Tom Cruise`
```

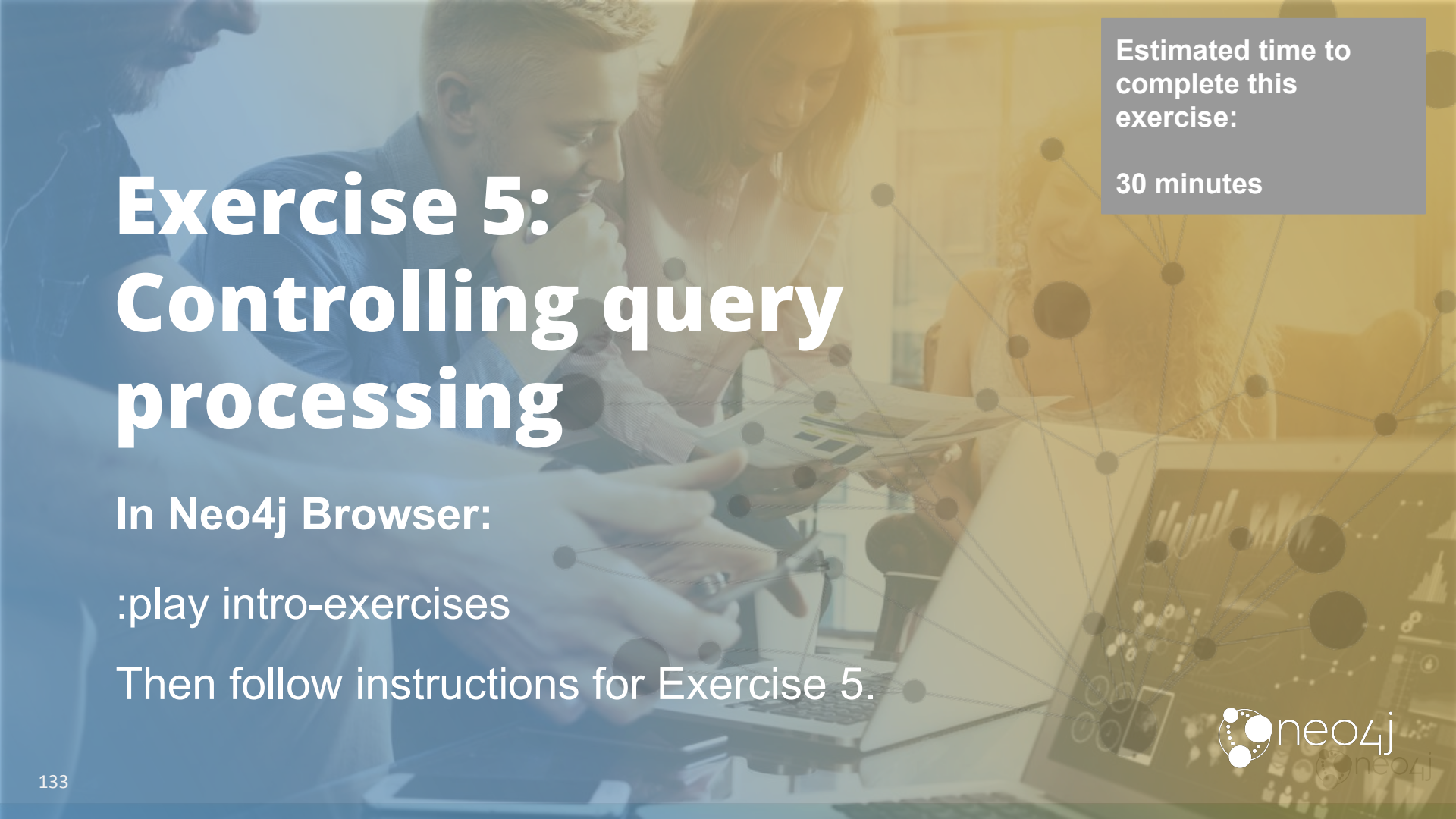


# Counting results

Find all of the actors and directors who worked on a movie, return the count of the number paths found between actors and directors and collect the movies as a list:

```
MATCH (actor:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(director:Person)
RETURN actor.name, director.name,  count(m) AS collaborations,
       collect(m.title) AS movies
```





# Exercise 5: Controlling query processing

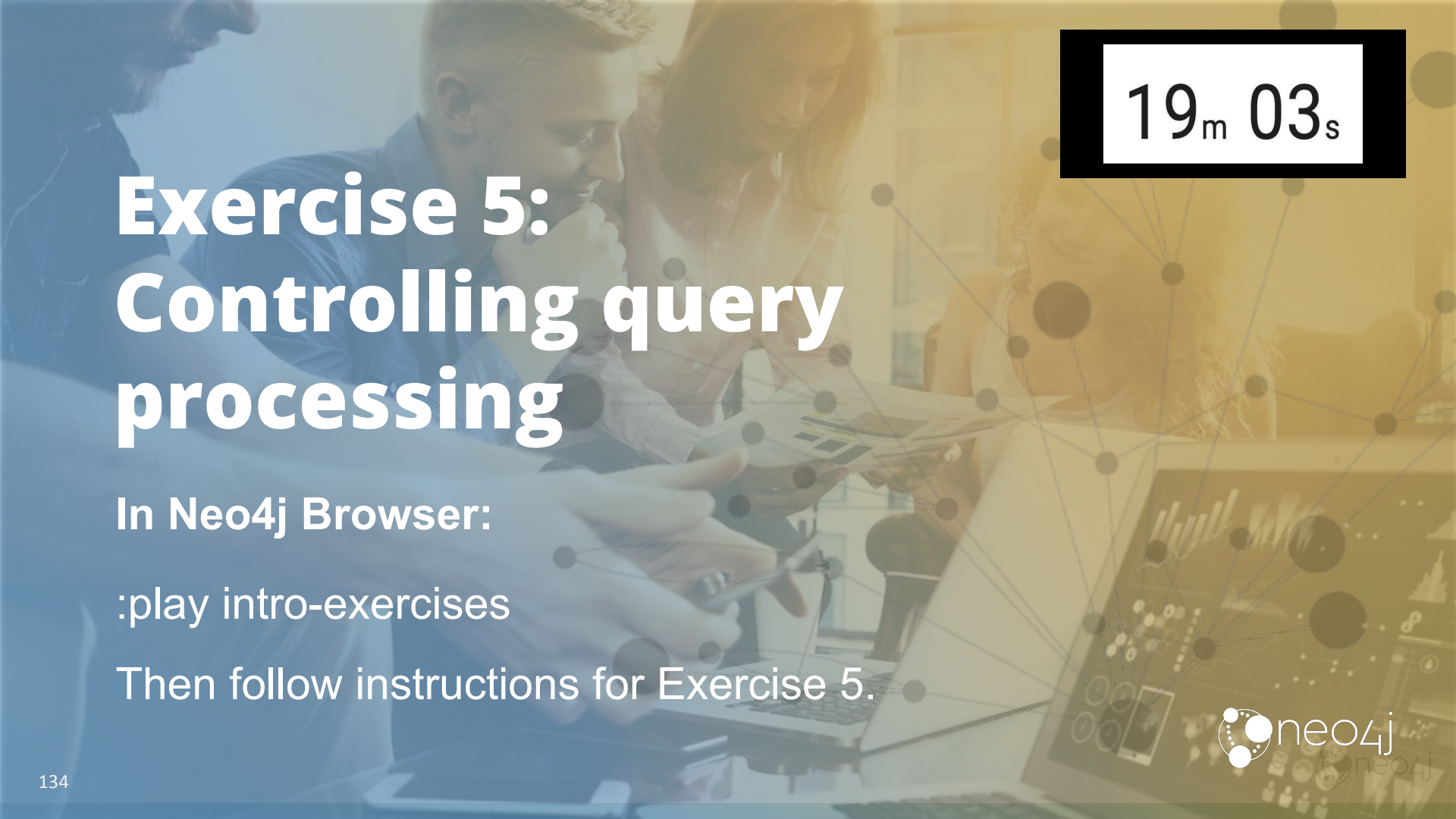
**Estimated time to complete this exercise:**

**30 minutes**

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 5.




# Exercise 5: Controlling query processing

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 5.


Check your understanding

# Question 1

Suppose you want to add a `WHERE` clause at the end of this statement to filter the results retrieved.

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

What variables, can you test in the `WHERE` clause:

Select the correct answers.

- ❑ p
- ❑ rel
- ❑ m
- ❑ PRODUCED

# Answer 1

Suppose you want to add a `WHERE` clause at the end of this statement to filter the results retrieved.

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

What variables, can you test in the `WHERE` clause:

Select the correct answers.

- ✔ p
- ✔ rel
- ✔ m
- ❑ PRODUCED

# Question 2

Suppose you want to retrieve all movies that have a *released* property value that is 2000, 2002, 2004, 2006, or 2008. Here is an incomplete Cypher example to return the *title* property values of all movies released in these years.

```
MATCH (m:MovMATCH (m:Movie)
WHERE m.released XX [2000, 2002, 2004, 2006, 2008]
RETURN m.title
```

What keyword do you specify for **XX**?

Select the correct answer.

- ❑ CONTAINS
- ❑ IN
- ❑ IS
- ❑ EQUALS

# Answer 2

Suppose you want to retrieve all movies that have a *released* property value that is 2000, 2002, 2004, 2006, or 2008. Here is an incomplete Cypher example to return the *title* property values of all movies released in these years.

```
MATCH (m:MovMATCH (m:Movie)
WHERE m.released XX [2000, 2002, 2004, 2006, 2008]
RETURN m.title
```

What keyword do you specify for **XX**?

Select the correct answer.

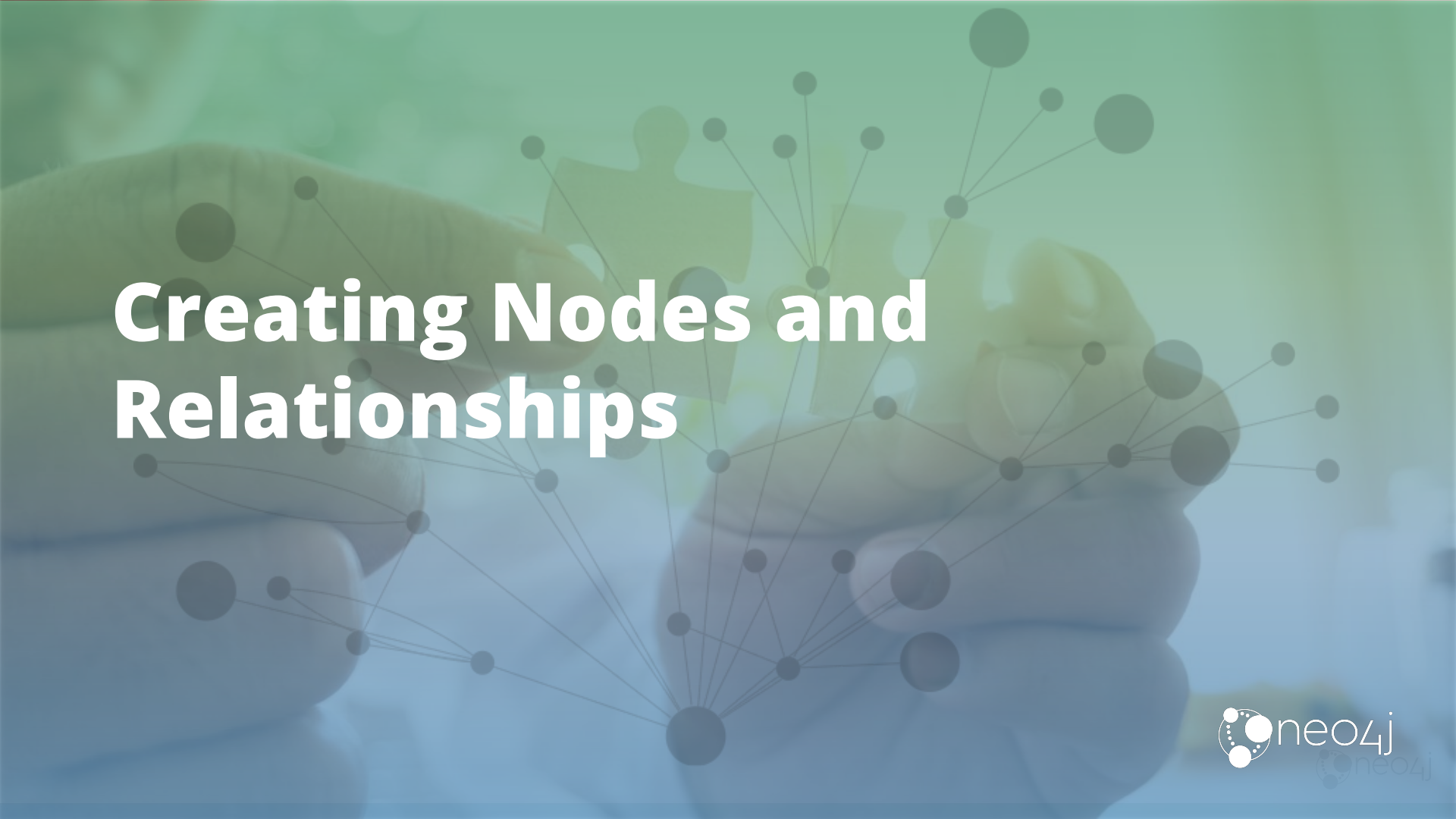
- ❏ CONTAINS
- ✔ IN
- ❏ IS
- ❏ EQUALS

# Summary

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

- Filter queries using the `WHERE` clause
- Control query processing
- Control what results are returned

# Creating Nodes and Relationships



# Overview

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

- Create a node:
  - Add and remove node labels.
  - Add and remove node properties.
  - Update properties.
- Create a relationship:
  - Add and remove properties for a relationship.
- Delete a node.
- Delete a relationship.
- Merge data in a graph:
  - Creating nodes.
  - Creating relationships.

# Creating a node

Create a node of type *Movie* with the *title* property set to *Batman Begins*:

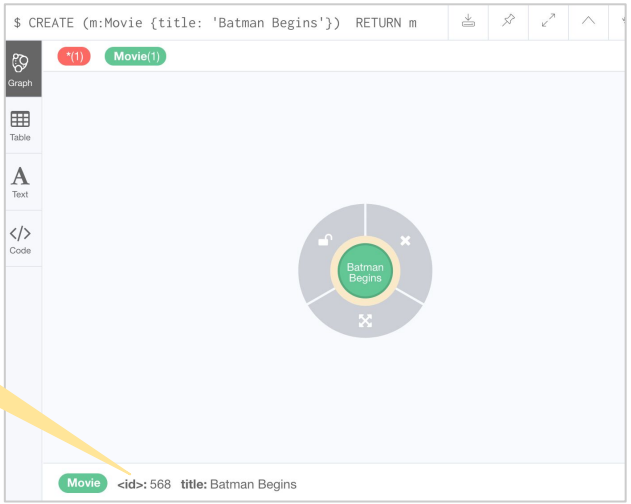
```
CREATE (:Movie {title: 'Batman Begins'})
```

Create a node of type *Movie* and *Action* with the *title* property set to *Batman Begins*:

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

Create a node of type *Movie* with the *title* property set to *Batman Begins* and return the node:

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

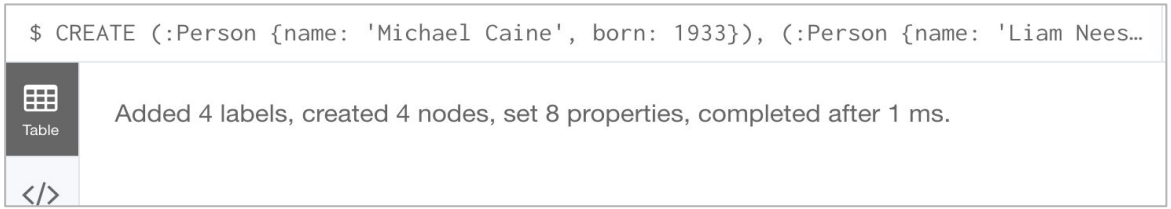



<id> is set by the graph engine

# Creating multiple nodes

Create some *Person* nodes for actors and the director for the movie, *Batman Begins*:

```
CREATE (:Person {name: 'Michael Caine', born: 1933}),
    (:Person {name: 'Liam Neeson', born: 1952}),
    (:Person {name: 'Katie Holmes', born: 1978}),
    (:Person {name: 'Benjamin Melniker', born: 1913})
```




**Important**: The graph engine will create a node with the same properties of a node that already exists. You can prevent this from happening in one of two ways:

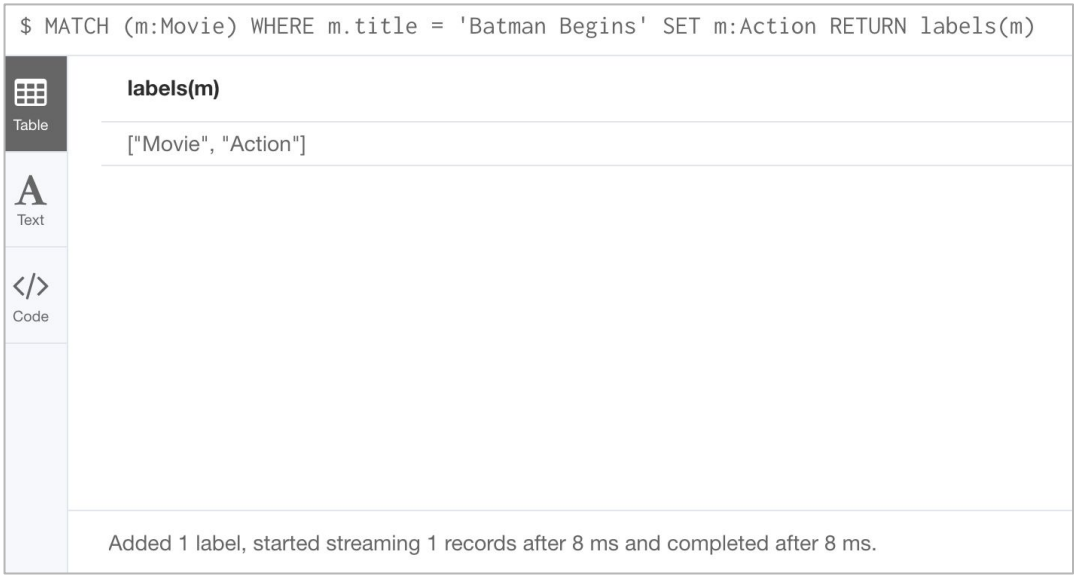
1. You can use `MERGE` rather than `CREATE` when creating the node.

2. You can add constraints to your graph.

# Adding a label to a node

Add the *Action* label to the movie, *Batman Begins*, return all labels for this node:

```
MATCH (m:Movie)
WHERE m.title = 'Batman Begins'
SET m:Action
RETURN labels(m)
```



# Removing a label from a node

Remove the *Action* label to the movie, *Batman Begins*, return all labels for this node:

```
MATCH (m:Movie:Action)
WHERE m.title = 'Batman Begins'
REMOVE m:Action
RETURN labels(m)
```

# Adding or updating properties for a node

- If property does not exist for the node, it is added with the specified value.
- If property exists for the node, it is updated with the specified value

Add the properties *released* and *lengthInMinutes* to the movie *Batman Begins*:

```
MATCH (m:Movie)
WHERE m.title = 'Batman Begins'
SET m.released = 2005, m.lengthInMinutes = 140
RETURN m
```

# Adding properties to a node - JSON style

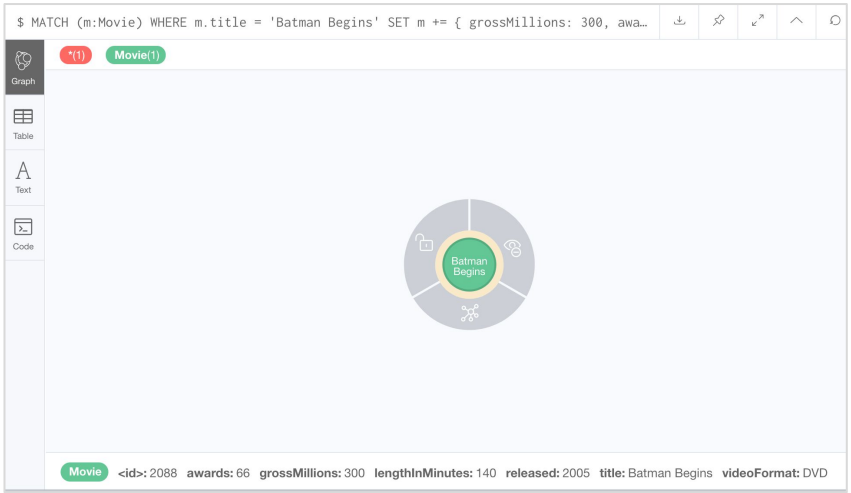
Add or update all properties: *title*, *released*, *lengthInMinutes*, *videoFormat*, and *grossMillions* for the movie *Batman Begins*:

```
MATCH (m:Movie)
WHERE m.title = 'Batman Begins'
SET  m = {title: 'Batman Begins',
          released: 2005,
          lengthInMinutes: 140,
          videoFormat: 'DVD',
          grossMillions: 206.5}
RETURN m
```

# Adding or updating properties for a node - JSON style

Add the *awards* property and update the *grossMillions* for the movie *Batman Begins*:

```
MATCH (m:Movie)
WHERE m.title = 'Batman Begins'
SET  m += { grossMillions: 300,
            awards: 66}
RETURN m
```



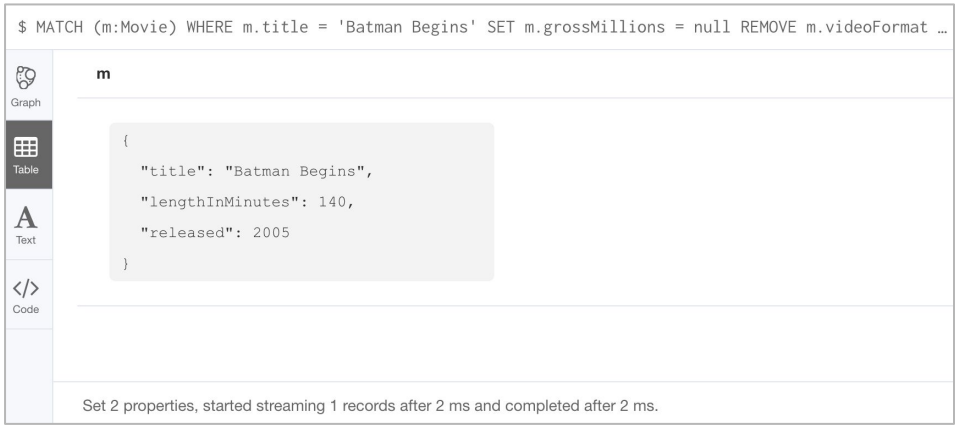
# Removing properties from a node

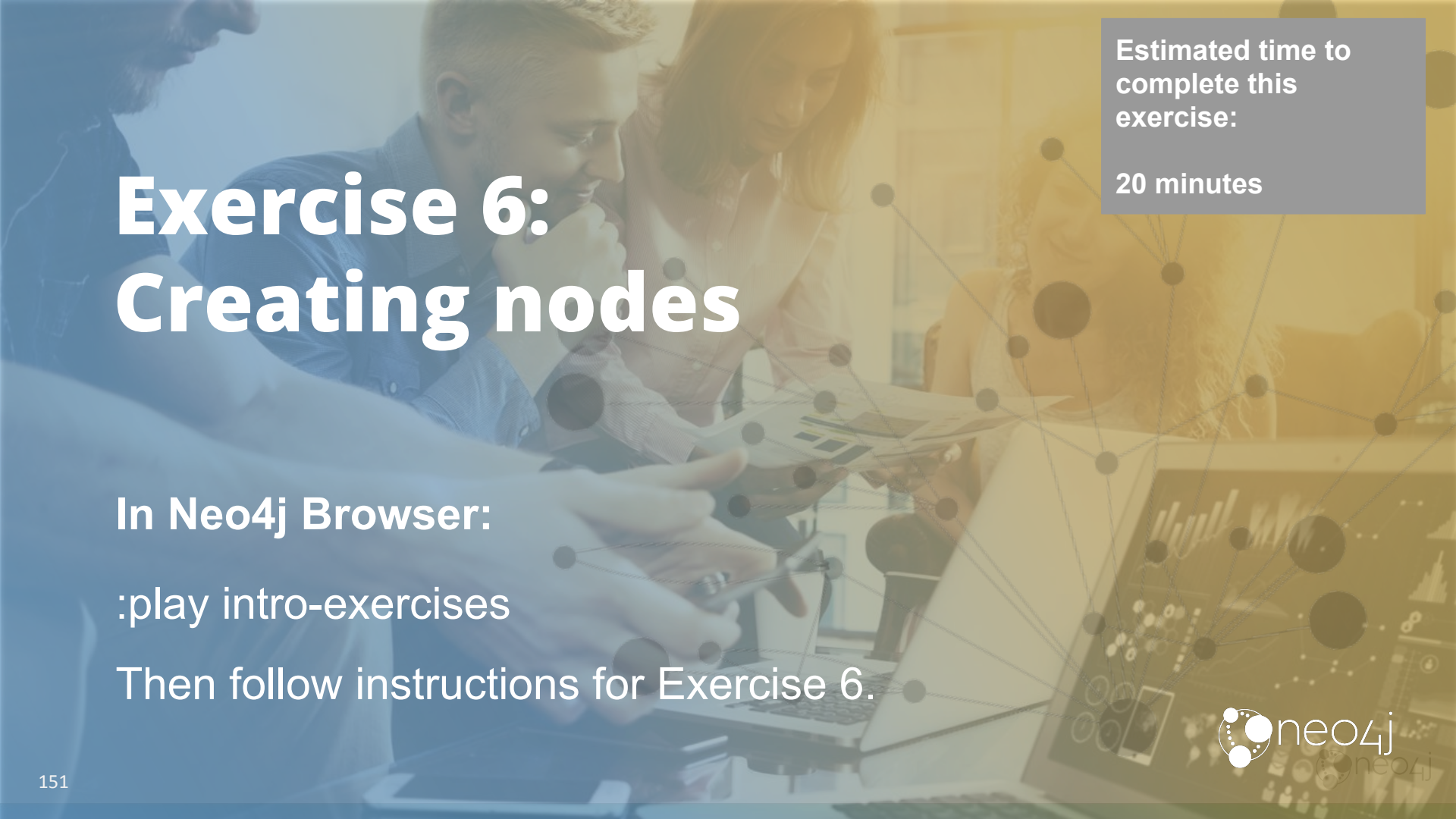
Properties can be removed in one of two ways:

- Set the property value to null
- Use the REMOVE keyword

Remove the grossMillions and videoFormat properties:

```
MATCH (m:Movie)
WHERE m.title = 'Batman Begins'
SET m.grossMillions = null
REMOVE m.videoFormat
RETURN m
```





# Exercise 6: Creating nodes

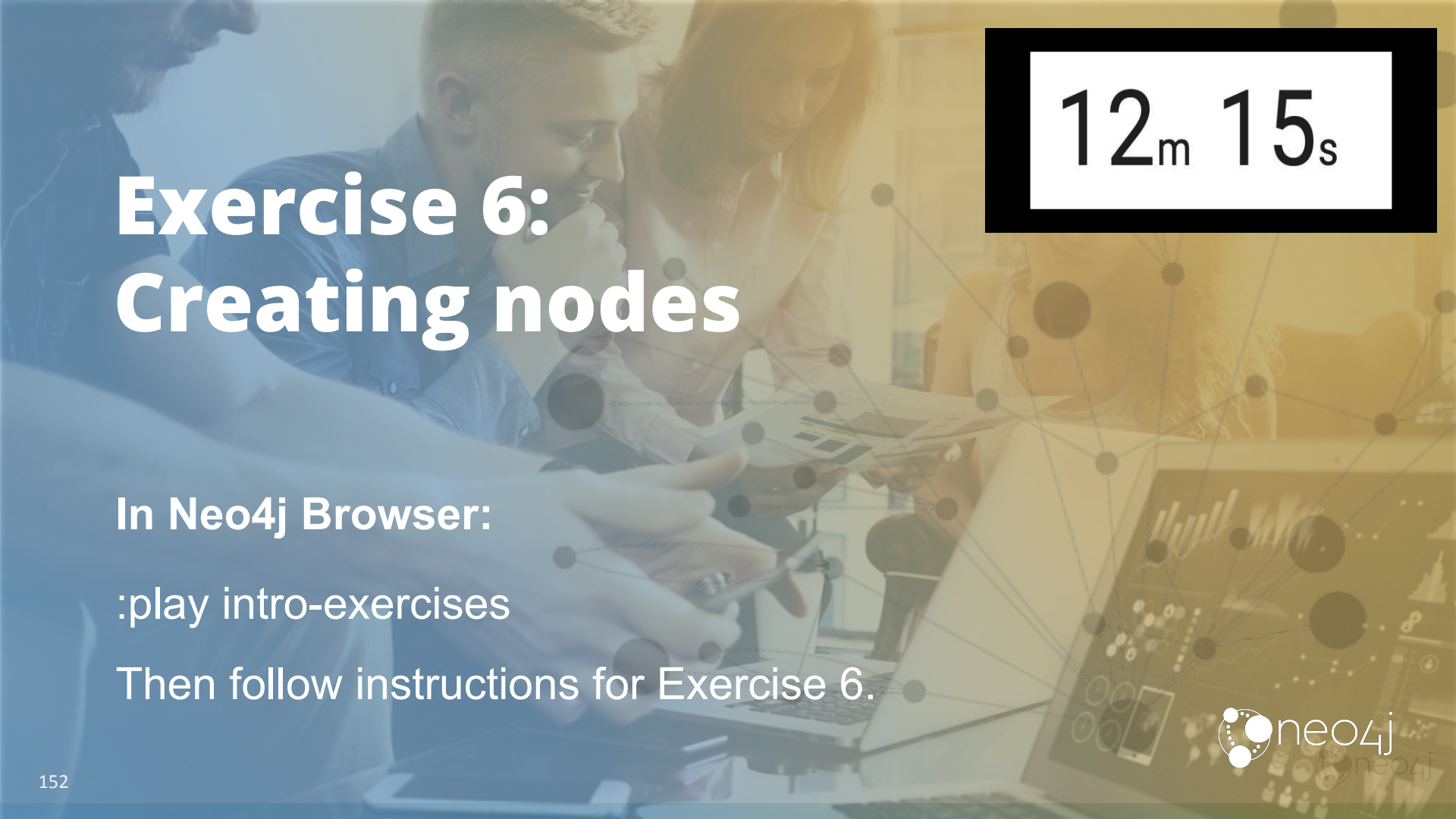
Estimated time to complete this exercise:

20 minutes

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 6.




# Exercise 6: Creating nodes

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 6.

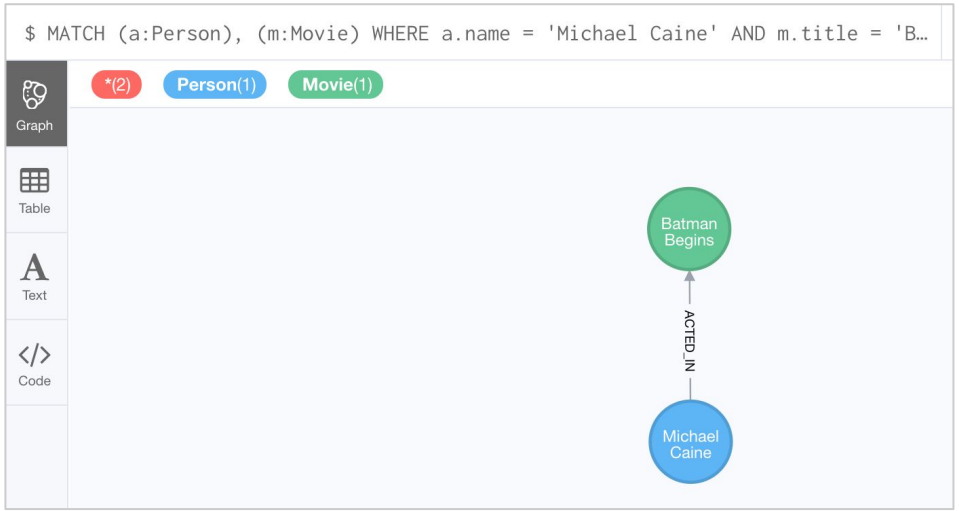
# Creating a relationship

You create a relationship by:

1. Finding the “from node”.
2. Finding the “to node”.
3. Using CREATE to add the directed relationship between the nodes.

Create the *:ACTED_IN* relationship between the *Person*, *Michael Caine* and the *Movie*, *Batman Begins*:

```
MATCH (a:Person), (m:Movie)
WHERE a.name = 'Michael Caine' AND
      m.title = 'Batman Begins'
CREATE (a)-[:ACTED_IN]->(m)
RETURN a, m
```



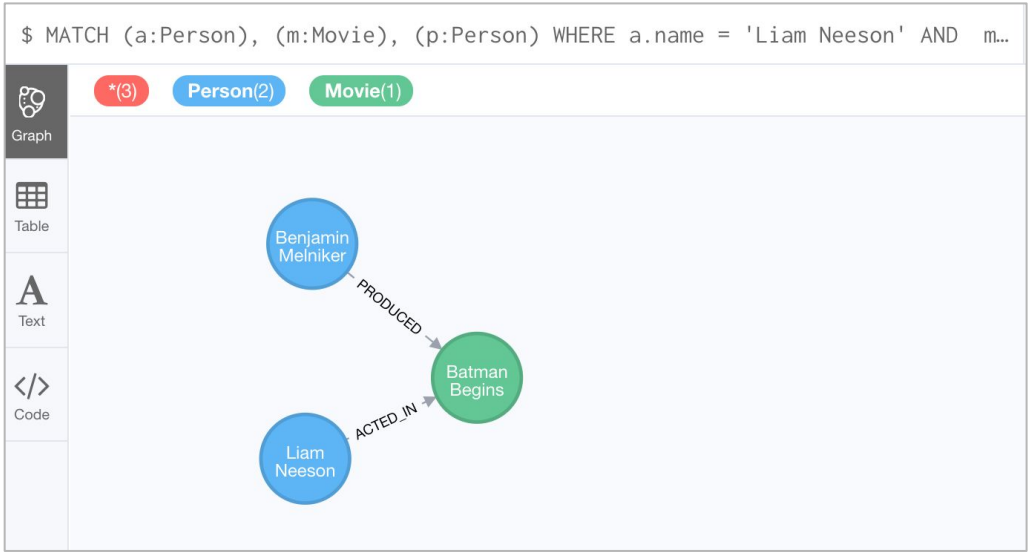



neo4j

# Creating multiple relationships

Create the *:ACTED_IN* relationship between the *Person*, *Liam Neeson* and the *Movie*, *Batman Begins and the :PRODUCED* relationship between the *Person*, *Benjamin Melniker* and same movie.

```
MATCH (a:Person), (m:Movie), (p:Person)
WHERE a.name = 'Liam Neeson' AND
      m.title = 'Batman Begins' AND
      p.name = 'Benjamin Melniker'
CREATE (a)-[:ACTED_IN]->(m)<-[:PRODUCED]-(p)
RETURN a, m, p
```





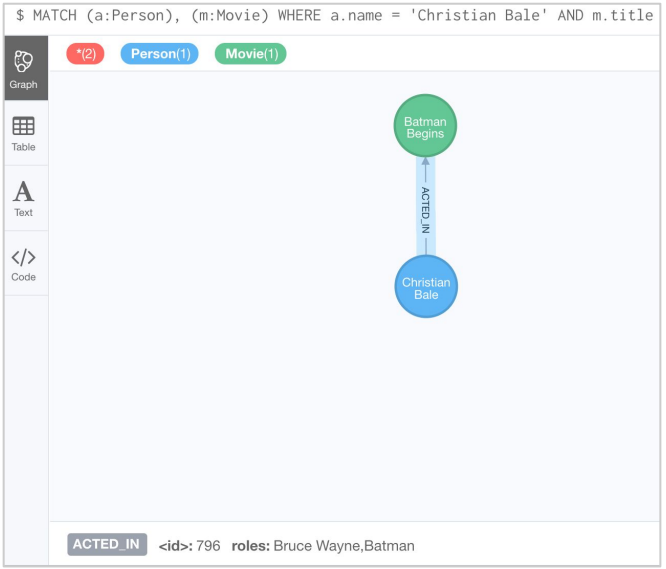

neo4j

# Adding properties to relationships

Same technique you use for creating and updating node properties.

Add the *roles* property to the *:ACTED_IN* relationship from Christian Bale to *Batman Begins*:

```
MATCH (a:Person), (m:Movie)
WHERE a.name = 'Christian Bale' AND
      m.title = 'Batman Begins' AND
      NOT exists((a)-[:ACTED_IN]->(m))
CREATE (a)-[rel:ACTED_IN]->(m)
SET rel.roles = ['Bruce Wayne','Batman']
RETURN a, m
```





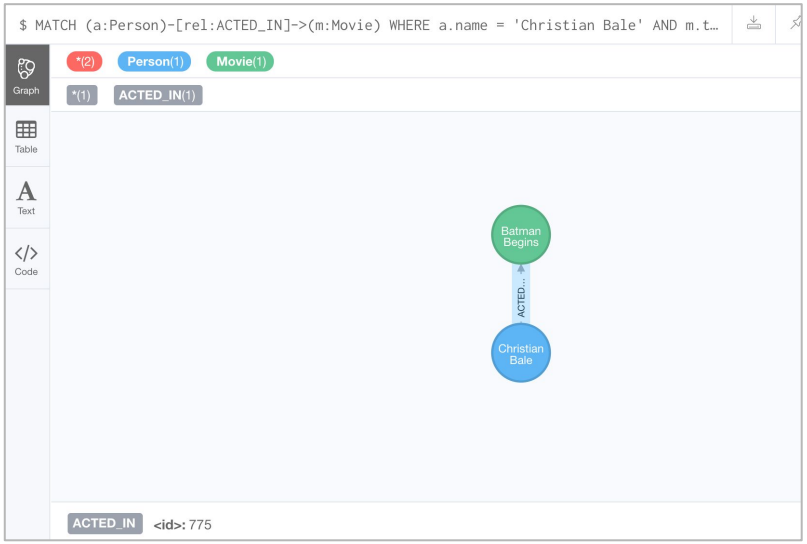

neo4j

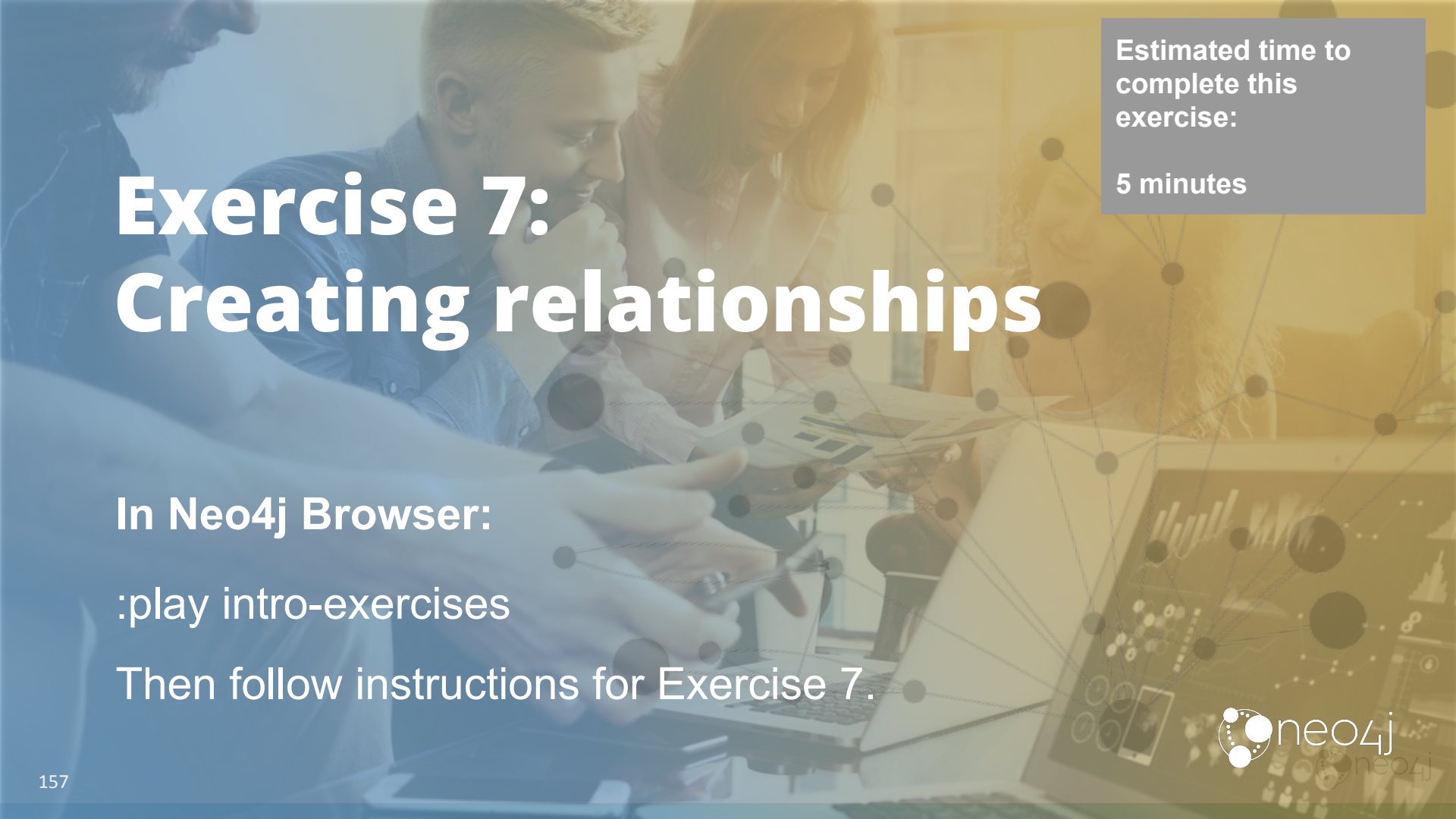
# Removing properties from relationships

Same technique you use for removing node properties.

Remove the *roles* property from the *:ACTED_IN* relationship from Christian Bale to *Batman Begins*:

```
MATCH (a:Person)-[rel:ACTED_IN]->(m:Movie)
WHERE a.name = 'Christian Bale' AND
      m.title = 'Batman Begins'
REMOVE rel.roles
RETURN a, rel, m
```





Estimated time to complete this exercise:

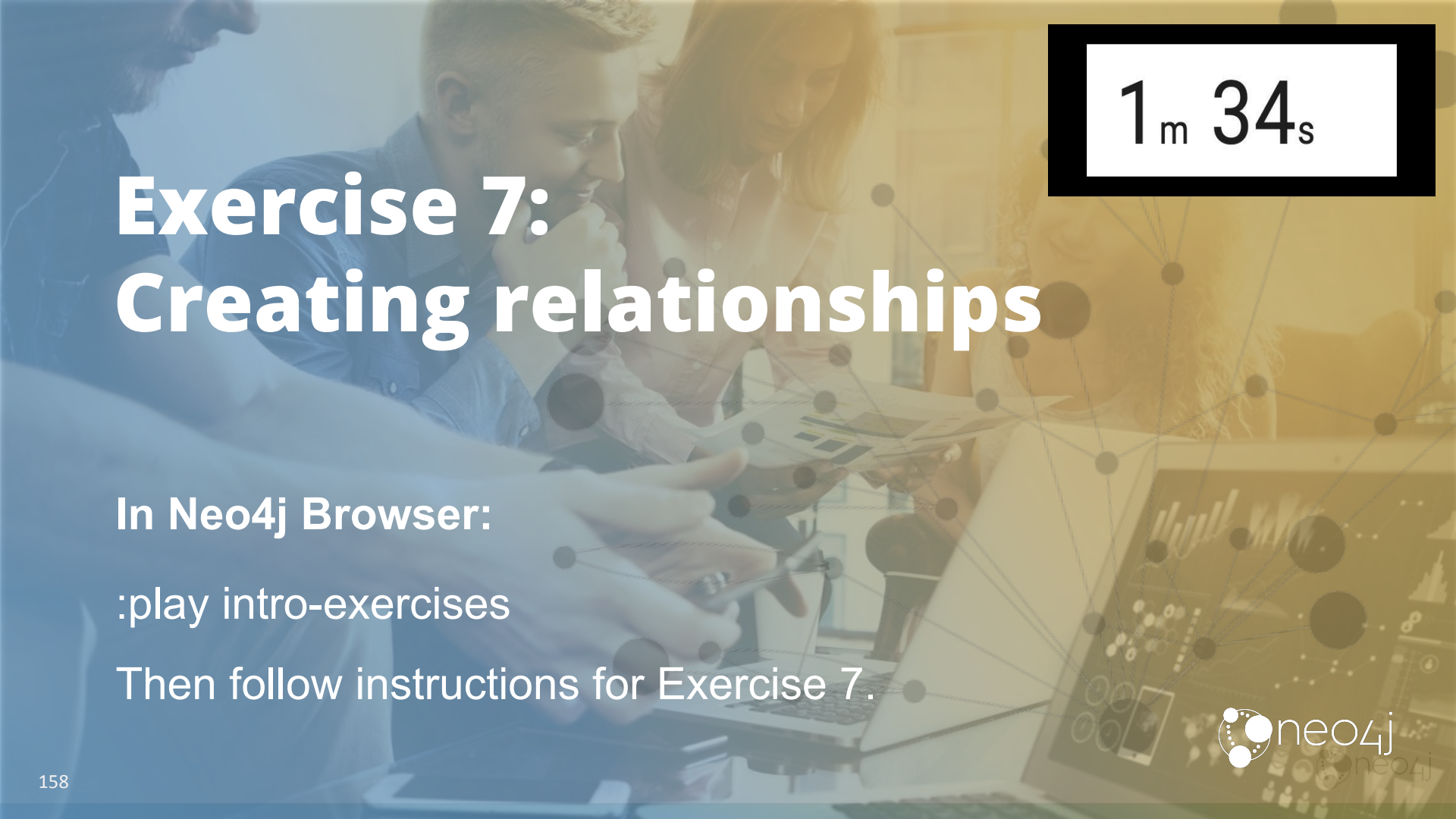
5 minutes

# Exercise 7: Creating relationships

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 7.




# Exercise 7: Creating relationships

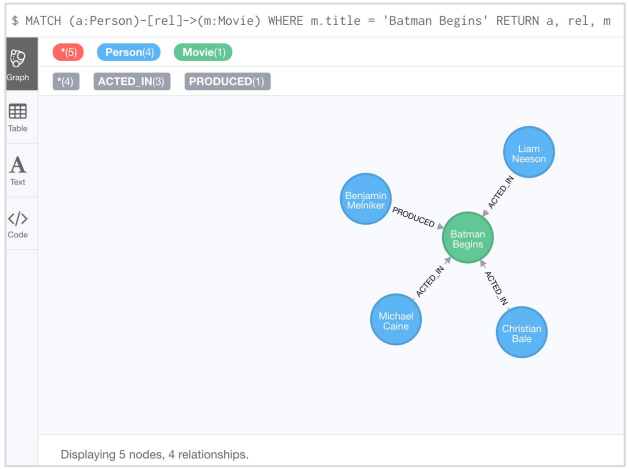
**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 7.

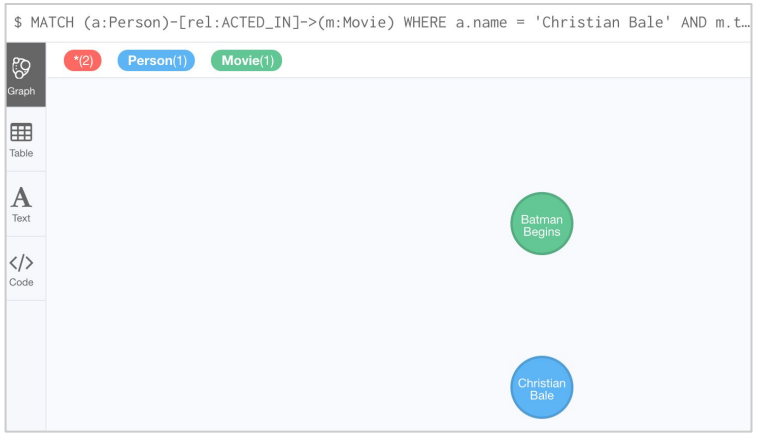
# Deleting a relationship

*Batman Begins* relationships:



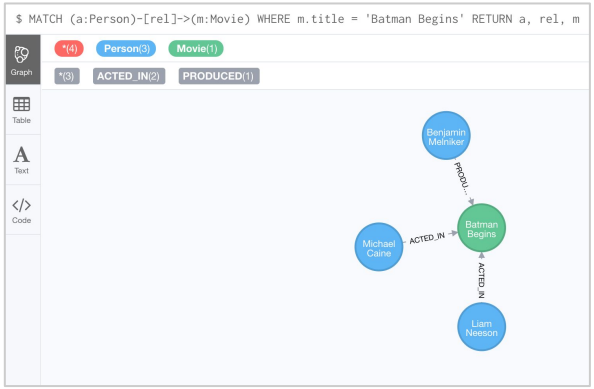

Delete the *:ACTED_IN* relationship between *Christian Bale* and *Batman Begins*:

```
MATCH (a:Person)-[rel:ACTED_IN]->(m:Movie)
WHERE a.name = 'Christian Bale' AND
      m.title = 'Batman Begins'
DELETE rel
RETURN a, m
```

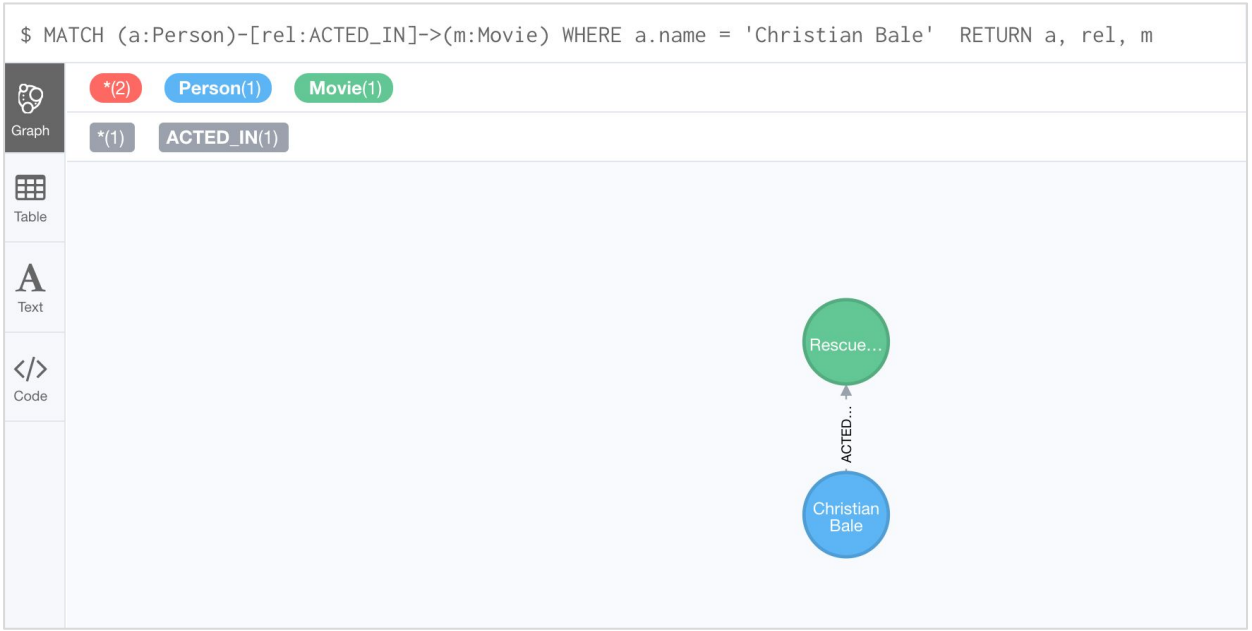


# After deleting the relationship from *Christian Bale* to *Batman Begins*

*Batman Begins* relationships:




*Christian Bale* relationships:

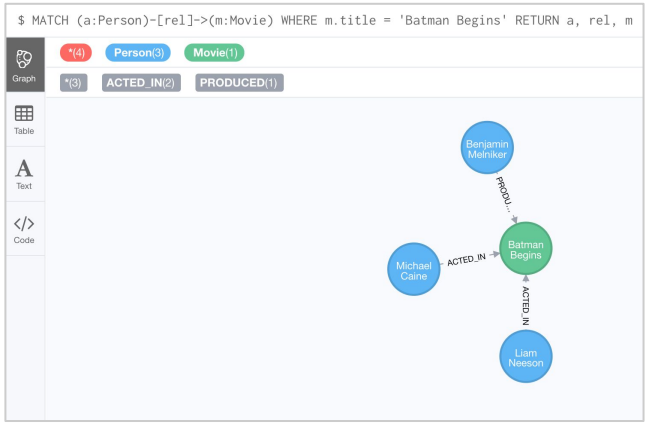





neo4j

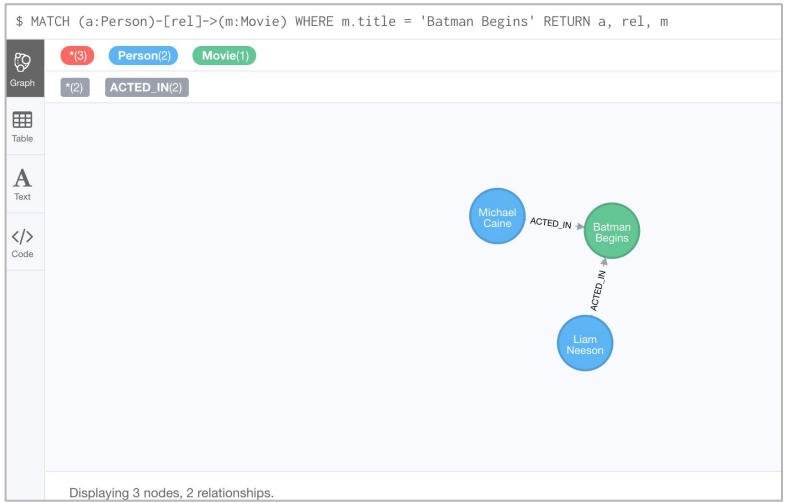
# Deleting a relationship and a node - 1

*Batman Begins* relationships:



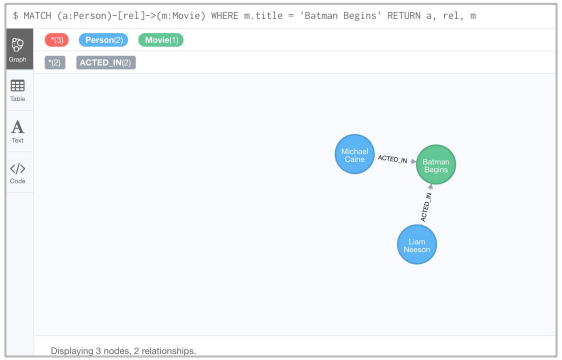

Delete the *:PRODUCED* relationship between *Benjamin Melniker* and *Batman Begins,* as well as the *Benjamin Melniker* node:

```
MATCH (p:Person)-[rel:PRODUCED]->(:Movie)
WHERE p.name = 'Benjamin Melniker'
DELETE rel, p
```



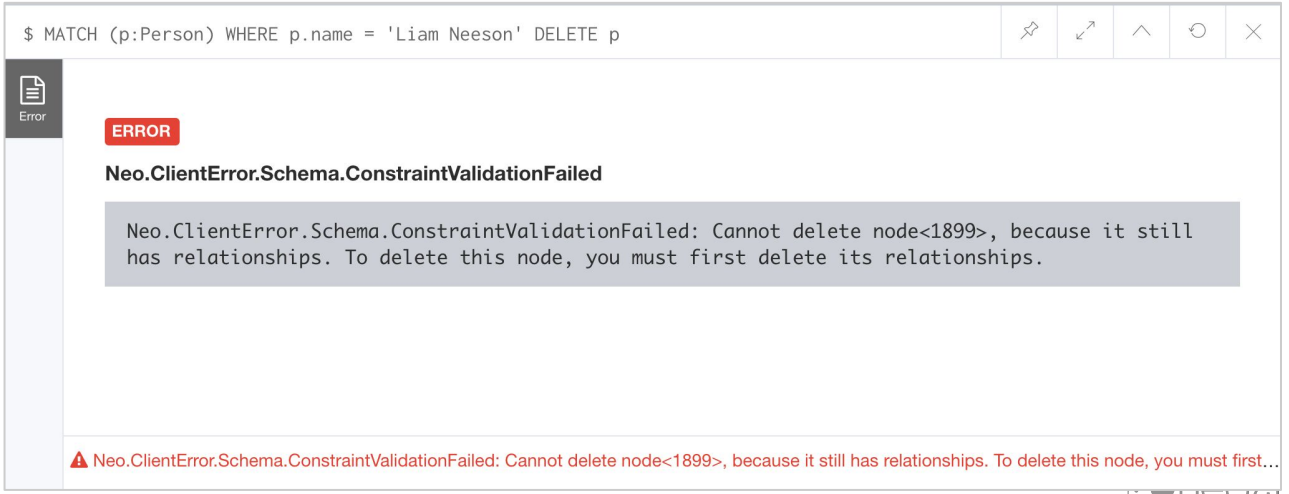
# Deleting a relationship and a node - 2

*Batman Begins* relationships:



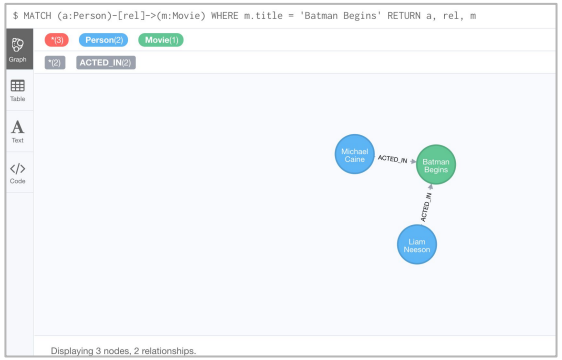

Attempt to delete *Liam Neeson* and not his relationships to any other nodes:

```
MATCH (p:Person)
WHERE p.name = 'Liam Neeson'
DELETE p
```



# Deleting a relationship and a node - 3

*Batman Begins* relationships:



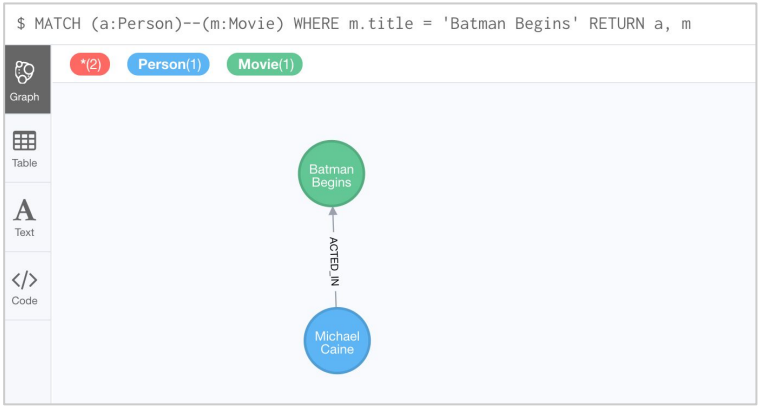

Delete *Liam Neeson* and his relationships to any other nodes:

```
MATCH (p:Person)
WHERE p.name = 'Liam Neeson'
DETACH DELETE p
```

```
$ MATCH (p:Person) WHERE p.name = 'Liam Neeson' DETACH DELETE p
```

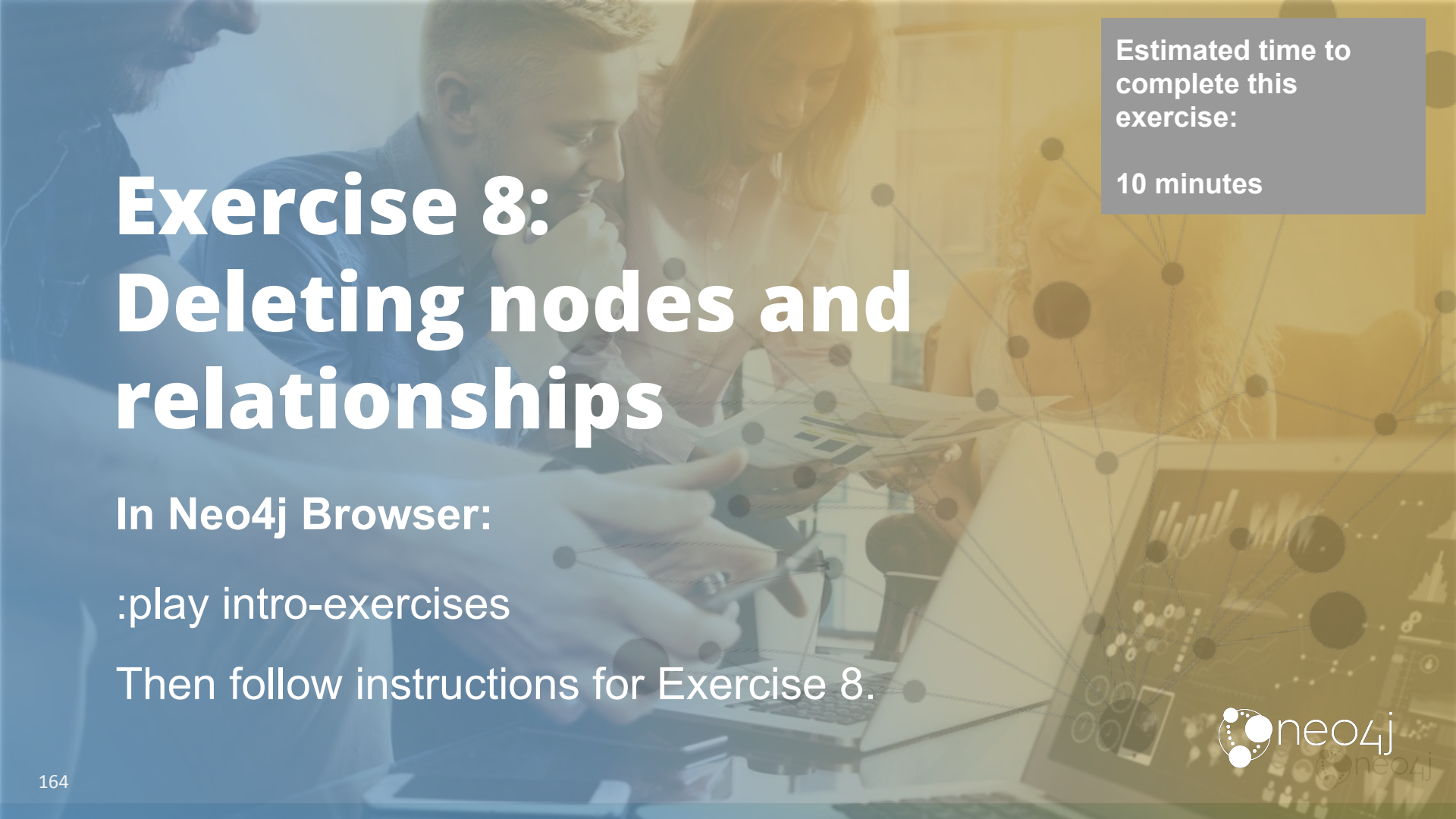
Table

Deleted 1 node, deleted 1 relationship, completed after 10 ms.






neo4j



Estimated time to complete this exercise:

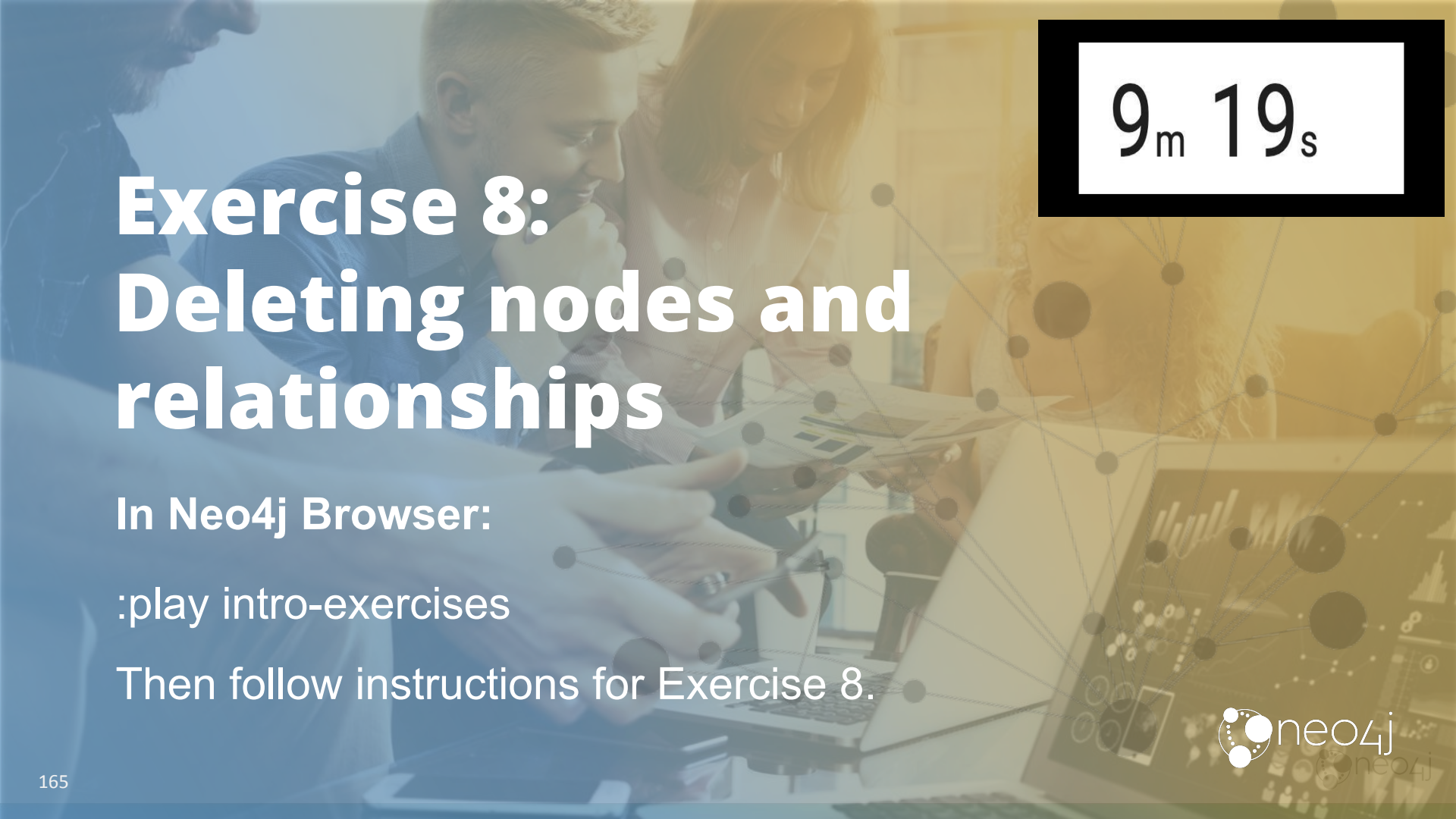
10 minutes

# Exercise 8: Deleting nodes and relationships

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 8.




# Exercise 8: Deleting nodes and relationships

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 8.

# Merging data in a graph

- Create a node with a different label (You do not want to add a label to an existing node.).
- Create a node with a different set of properties (You do not want to update a node with existing properties.).
- Create a unique relationship between two nodes.

# Using MERGE to create nodes

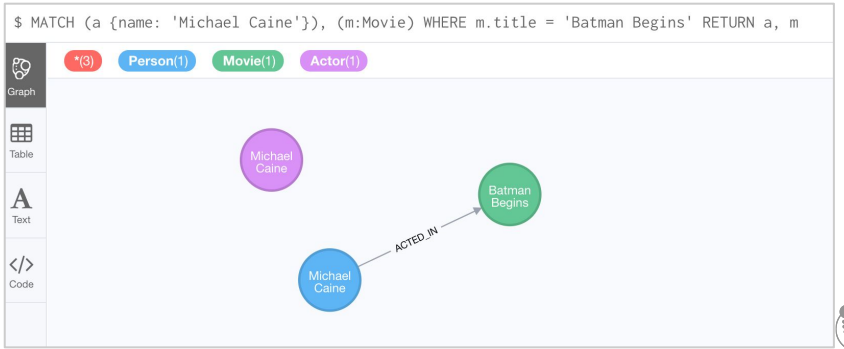
Current *Michael Caine Person* node:


Add a *Michael Caine Actor* node with a value of *1933* for *born* using MERGE. The *Actor* node is not found so a new node is created:

```
MERGE (a:Actor {name: 'Michael Caine'})
SET a.born=1933
RETURN a
```


**Important**: Only specify properties that will have unique keys when you merge.

Resulting *Michael Caine nodes*:



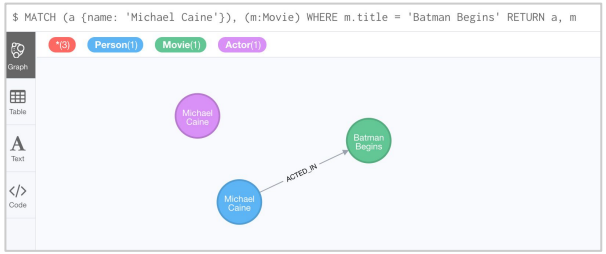
# Using MERGE to create relationships

Add the relationship(s) from all *Person* nodes with a *name* property that ends with *Caine* to the *Movie* node, *Batman Begins*:

```
MATCH (p:Person), (m:Movie)
WHERE m.title = 'Batman Begins' AND
p.name ENDS WITH 'Caine'
MERGE (p)-[:ACTED_IN]->(m)
RETURN p, m
```

# Specifying creation behavior for the merge

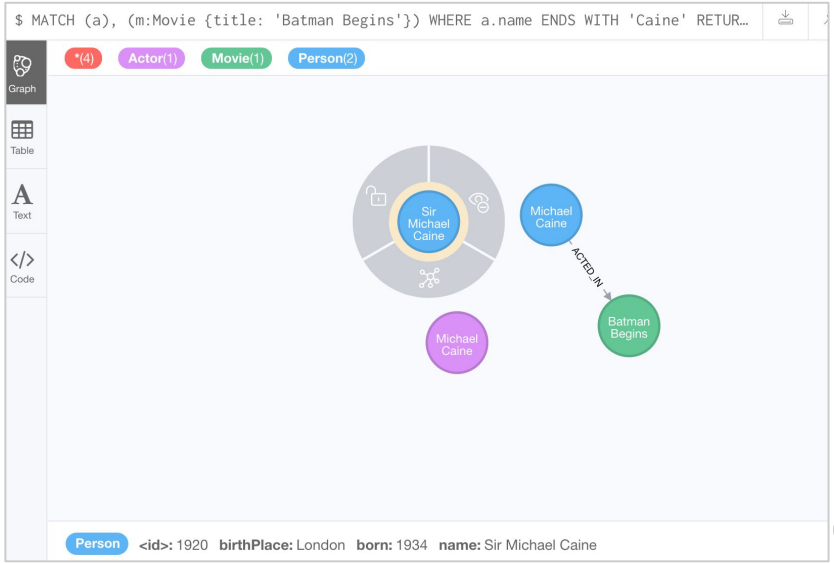
Current *Michael Caine* nodes:




Add a *Sir Michael Caine Person* node with a *born* value of *1934* for *born* using MERGE and also set the *birthPlace* property:

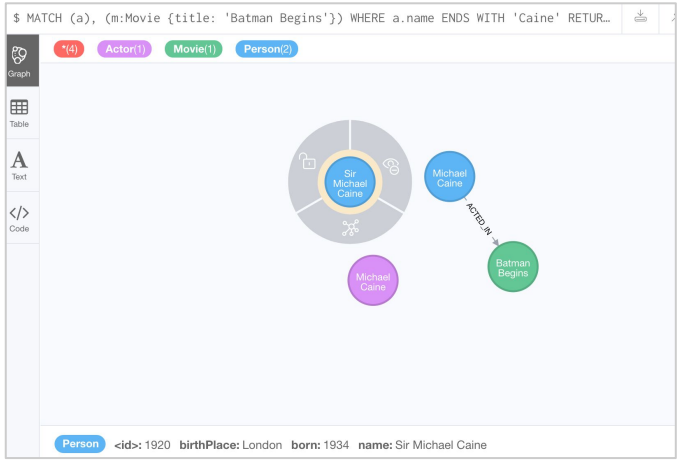
```
MERGE (a:Person {name: 'Sir Michael Caine'})
ON CREATE SET a.born = 1934,
              a.birthPlace = 'London'
RETURN a
```

Resulting *Michael Caine nodes*:



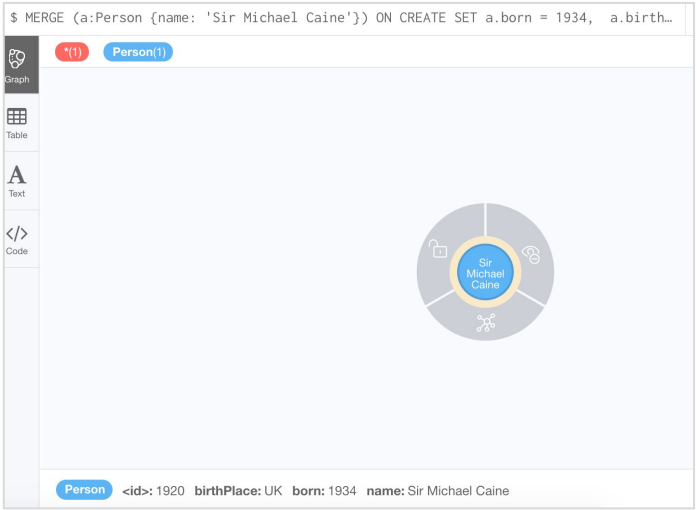
# Specifying match behavior for the merge

Current *Michael Caine* nodes:




Add or update the *Michael Caine Person* node:

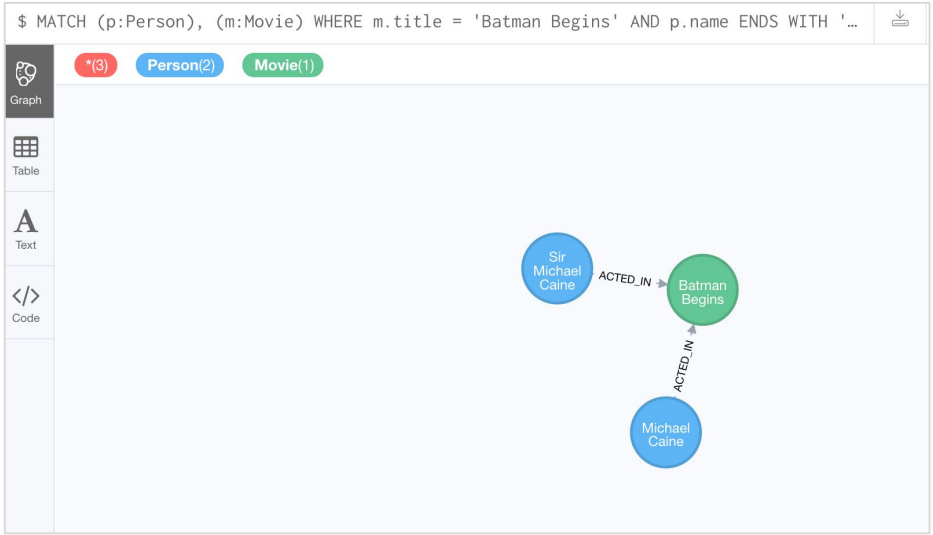
```
MERGE (a:Person {name: 'Sir Michael Caine'})
ON CREATE SET a.born = 1934,
              a.birthPlace = 'UK'
ON MATCH SET a.birthPlace = 'UK'
RETURN a
```

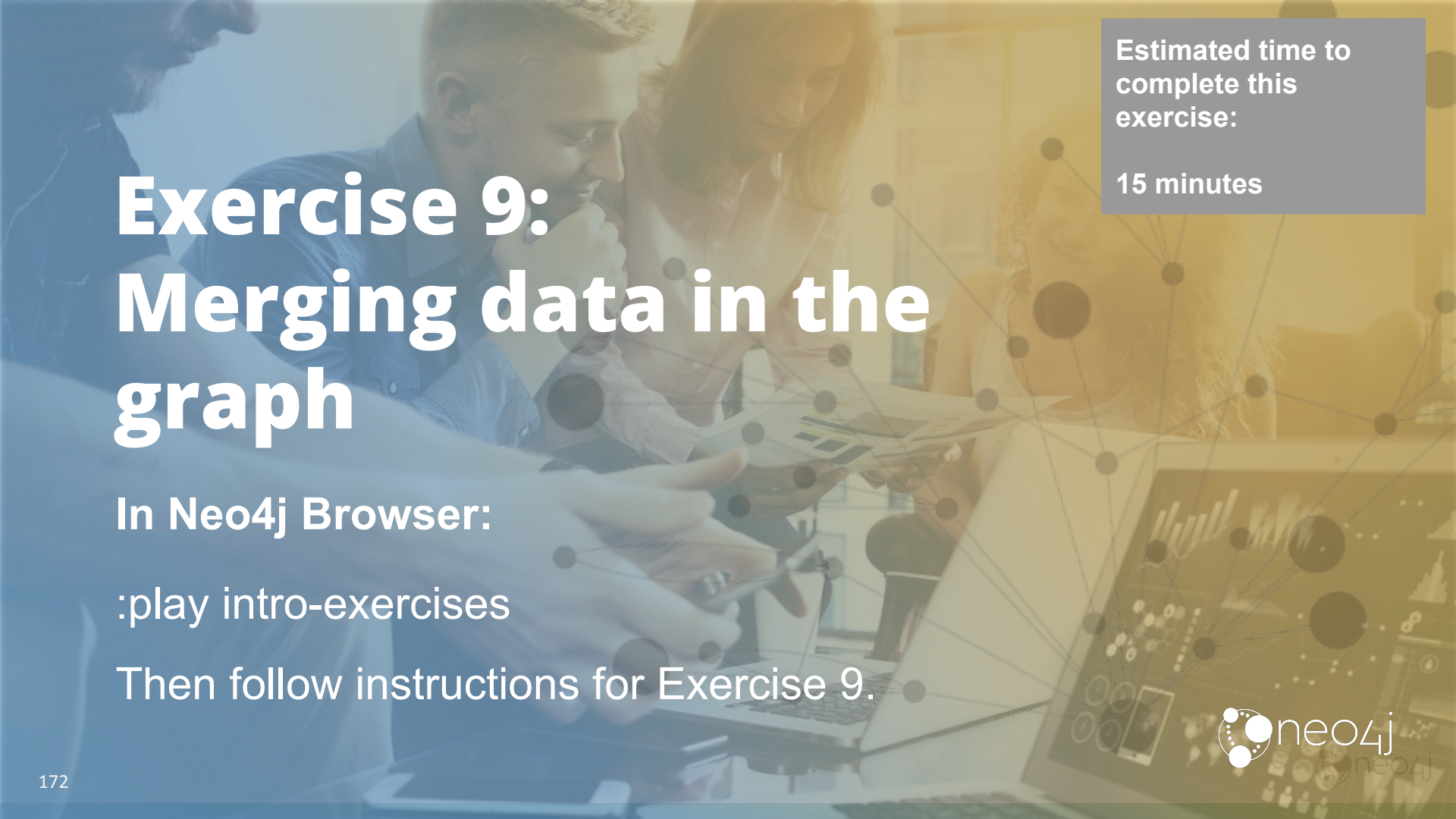


# Using MERGE to create relationships

Make sure that all *Person* nodes with a person whose name ends with *Caine* are connected to the *Movie* node, *Batman Begins*.

```
MATCH (p:Person), (m:Movie)
WHERE m.title = 'Batman Begins' AND p.name ENDS WITH 'Caine'
MERGE (p)-[:ACTED_IN]->(m)
RETURN p, m
```





Estimated time to complete this exercise:

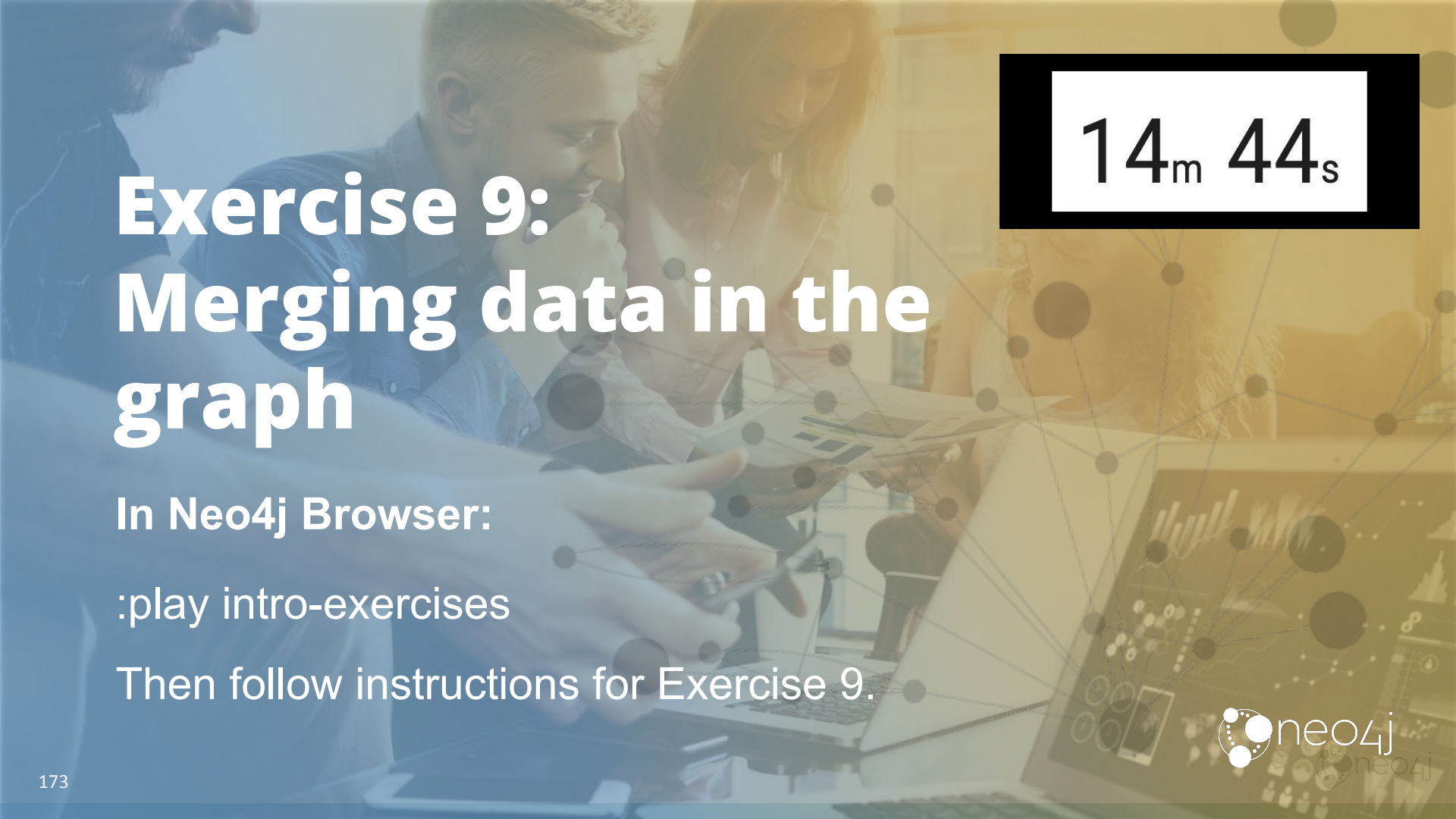
15 minutes

# Exercise 9: Merging data in the graph

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 9.




# Exercise 9: Merging data in the graph

**In Neo4j Browser:**

:play intro-exercises

Then follow instructions for Exercise 9.

# Check your understanding

# Question 1

What Cypher clauses can you use to create a node?

Select the correct answers.

- ❑ CREATE
- ❑ CREATE NODE
- ❑ MERGE
- ❑ ADD

# Answer 1

What Cypher clauses can you use to create a node?

Select the correct answers.

✔ CREATE

❑ CREATE NODE

✔ MERGE

❑ ADD

# Question 2

Suppose that you have retrieved a node, *s* with a property, *color*:

What Cypher clause do you add here to delete the *color* property from this node?

Select the correct answers.

- ❏ DELETE s.color
- ❏ SET s.color=null
- ❏ REMOVE s.color
- ❏ SET s.color=?

# Answer 2

Suppose that you have retrieved a node, *s* with a property, *color*:

What Cypher clause do you add here to delete the *color* property from this node?

Select the correct answers.

- ❏ DELETE s.color
- ✔ SET s.color=null
- ✔ REMOVE s.color
- ❏ SET s.color=?

# Question 3

Suppose you retrieve a node, *n* in the graph that is related to other nodes. What Cypher clause do you write to delete this node and its relationships in the graph?

Select the correct answers.

- ❏ DELETE n
- ❏ DELETE n WITH RELATIONSHIPS
- ❏ REMOVE n
- ❏ DETACH DELETE n

# Answer 3

Suppose you retrieve a node, *n* in the graph that is related to other nodes. What Cypher clause do you write to delete this node and its relationships in the graph?

Select the correct answers.

- ❏ DELETE n
- ❏ DELETE n WITH RELATIONSHIPS
- ❏ REMOVE n
- ✔ DETACH DELETE n

# Summary

You should be able to write Cypher statements to:

- Create a node:
  - Add and remove node labels.
  - Add and remove node properties.
  - Update properties.
- Create a relationship:
  - Add and remove properties for a relationship.
- Delete a node.
- Delete a relationship.
- Merge data in a graph:
  - Creating nodes.
  - Creating relationships.

# 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/.