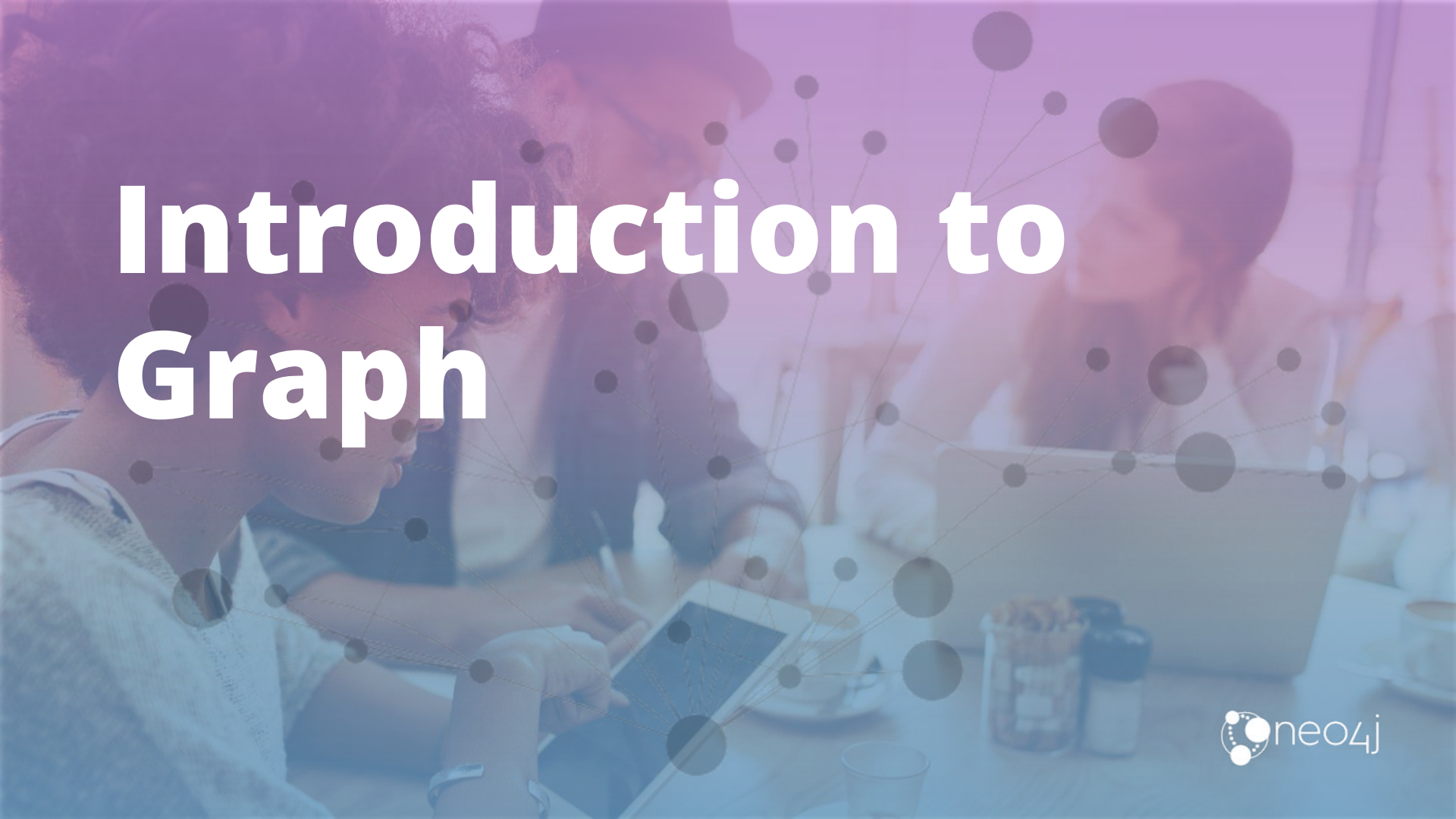
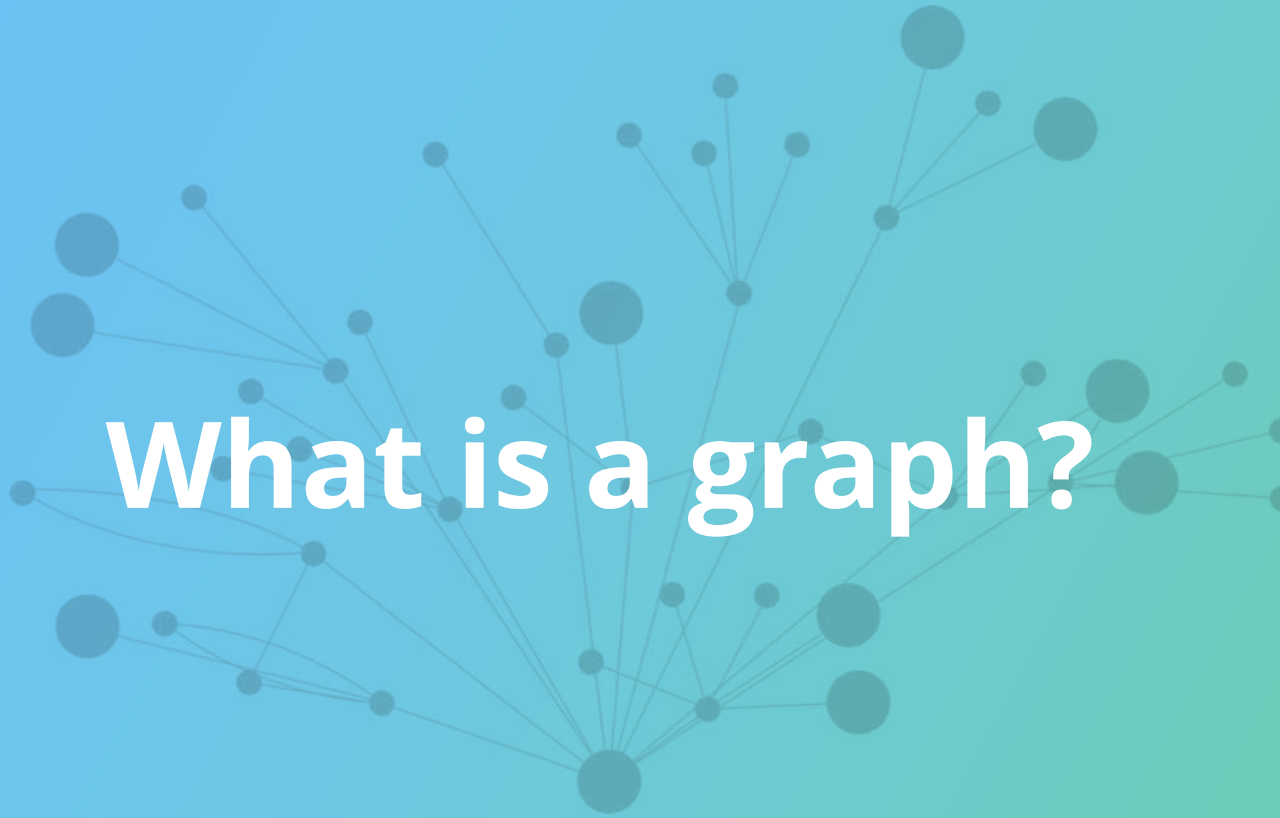
Introduction to Neo4j

Soham Dhodapkar
soham@neo4j.com

neo4j

Introduction to Graph



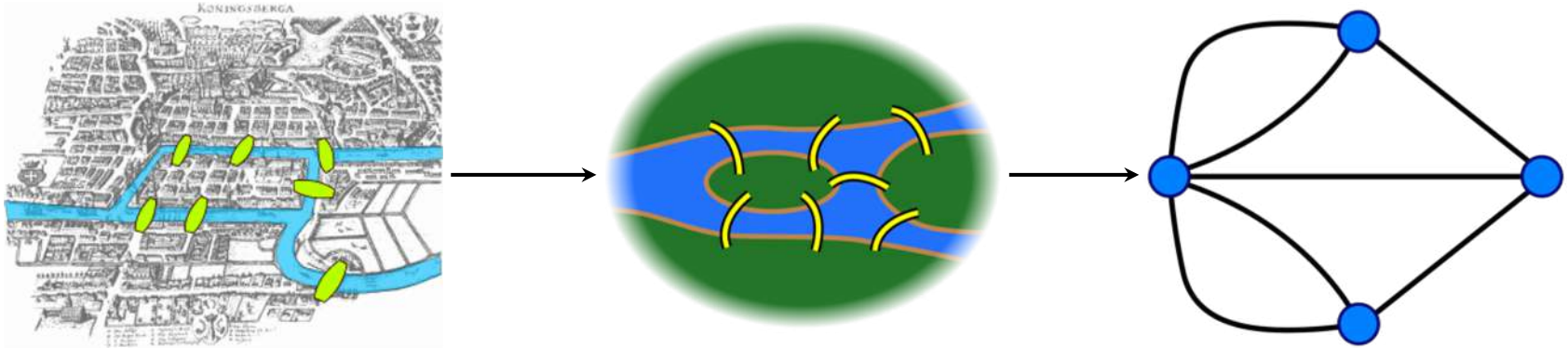


What is a graph?

neo4j

A graph is...

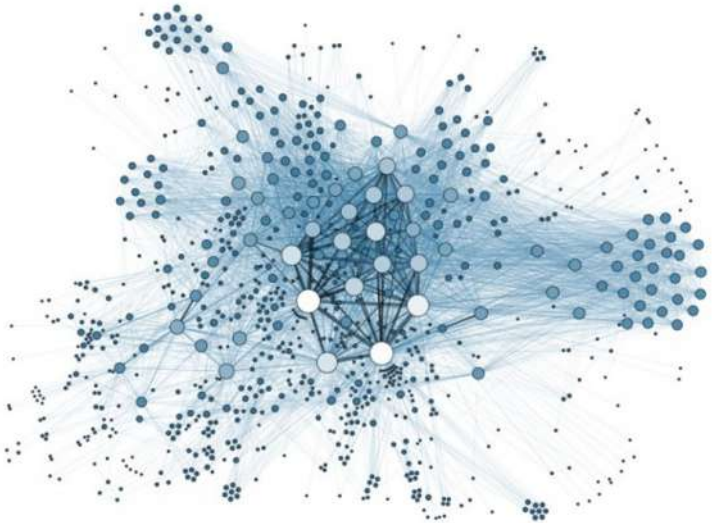
...a set of discrete objects, each of which has some set of relationships with the other objects



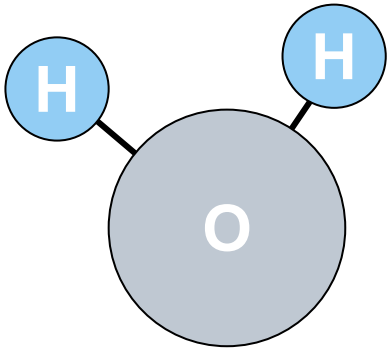
Seven Bridges of Konigsberg problem. Leonhard Euler, 1735

Anything can be a graph

the Internet



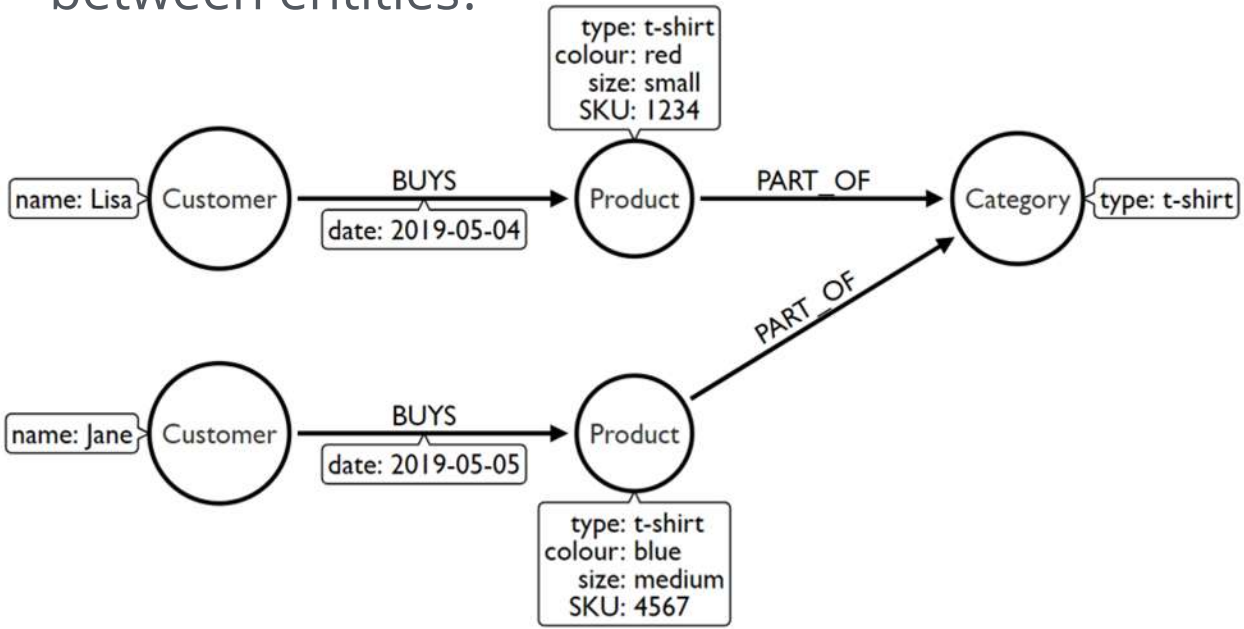
a water molecule



Scenarios for identifying graph-shaped problems

Identifying graph-shaped problems

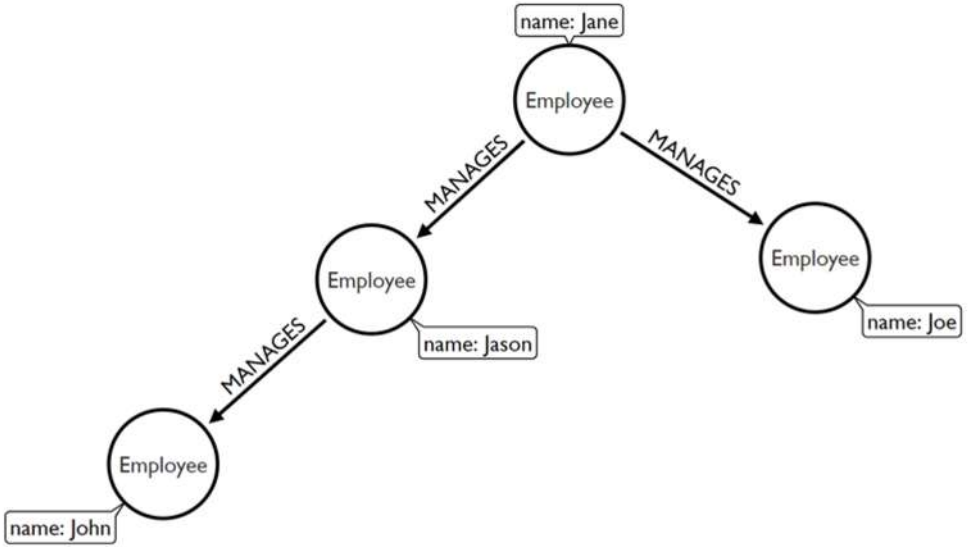
Scenario 1: Does our problem involve understanding relationships between entities?



- Recommendations
- Next best action
- Fraud detection
- Identity resolution
- Data lineage

Identifying graph-shaped problems

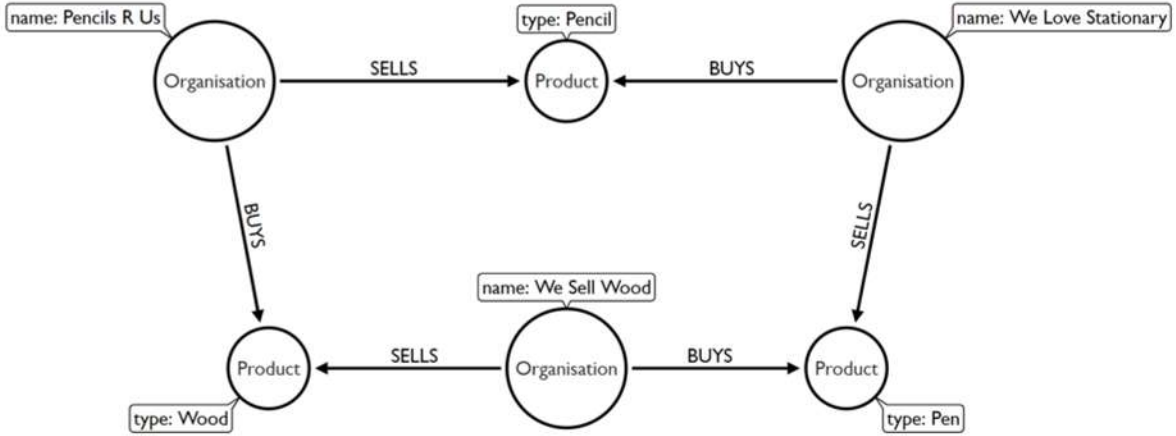
Scenario 2: Does the problem involve a lot of self-referencing to the same type of entity?



- Organisational hierarchies
- Social influencers
- Friends of friends
- Churn detection

Identifying graph-shaped problems

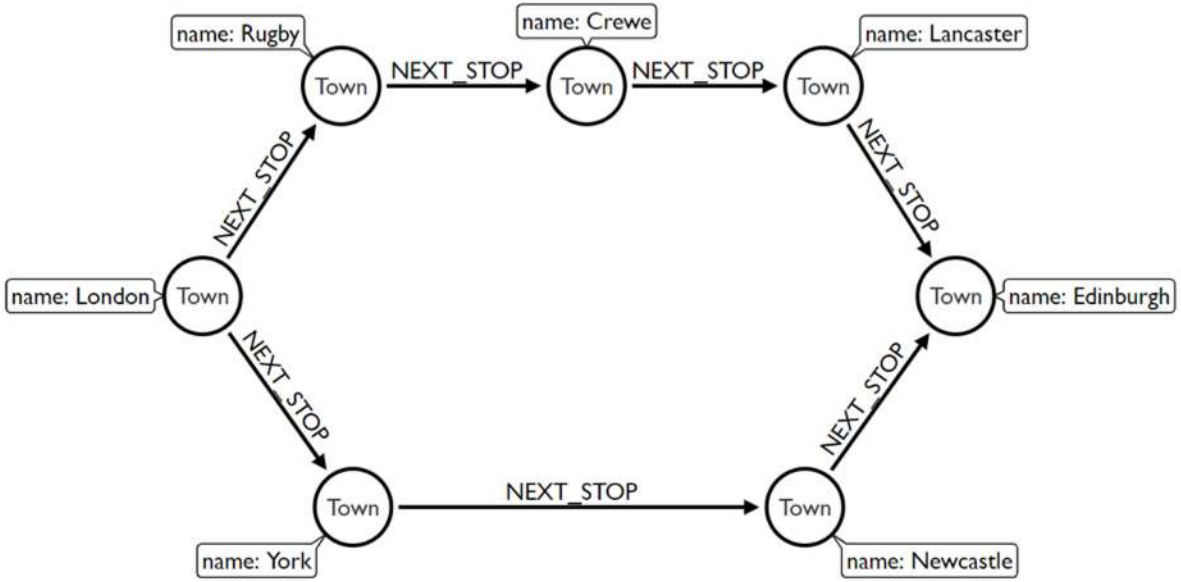
Scenario 3: Does the problem explore relationships of varying or unknown depth?



- Supply chain visibility
- Bill of Materials
- Network management

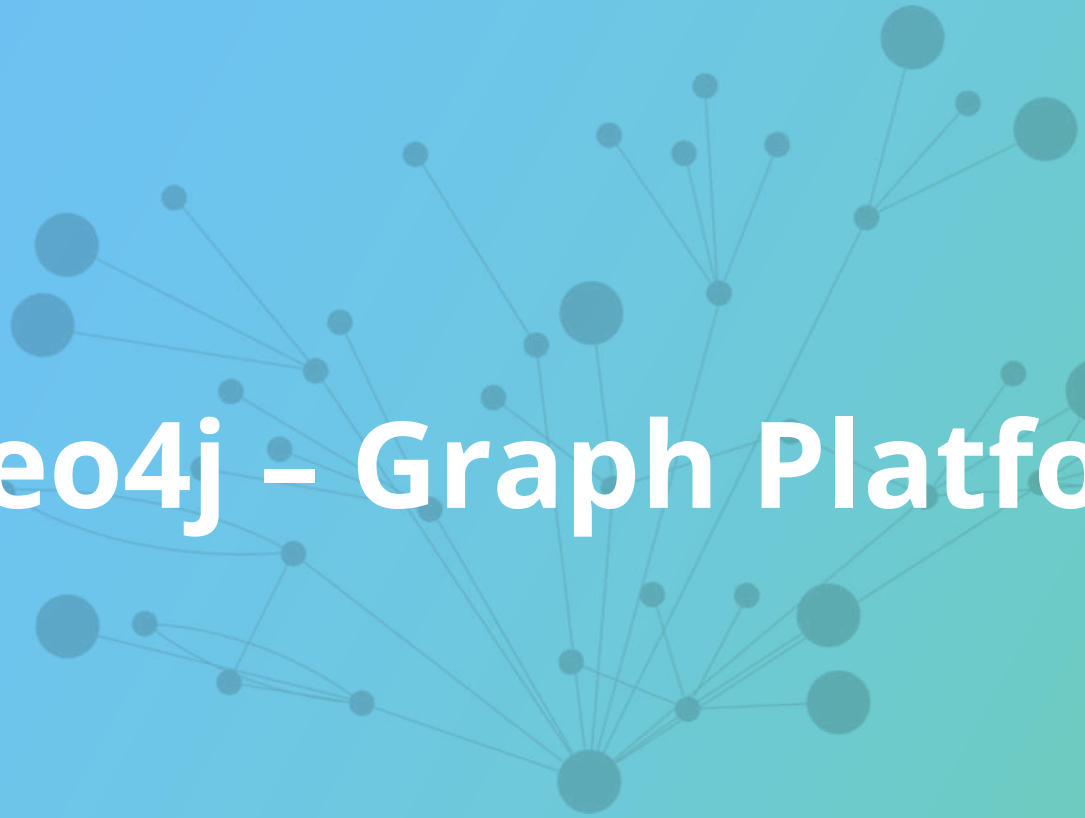
Identifying graph-shaped problems

Scenario 4: Does our problem involve discovering lots of different routes or paths?



- Logistics and routing
- Infrastructure management
- Dependency tracing

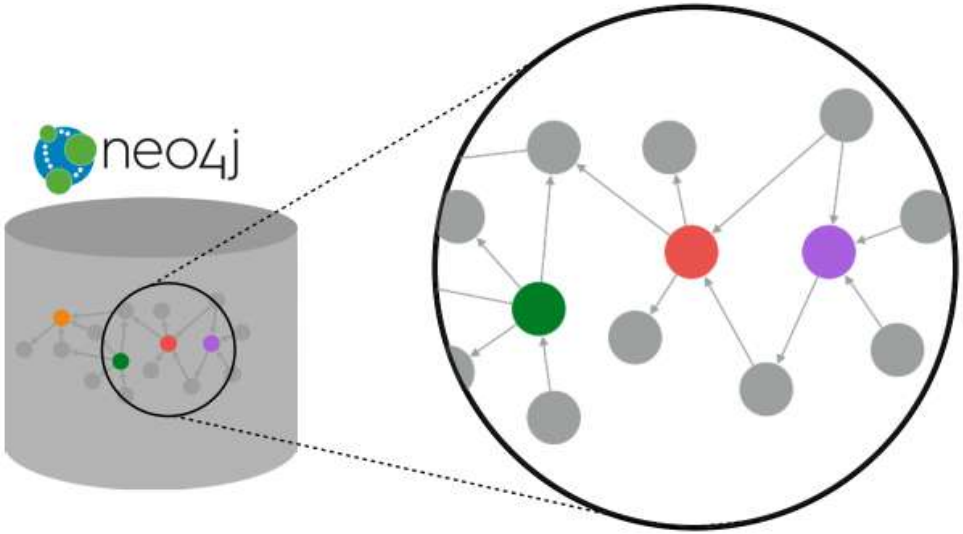
Neo4j – Graph Platform



neo4j

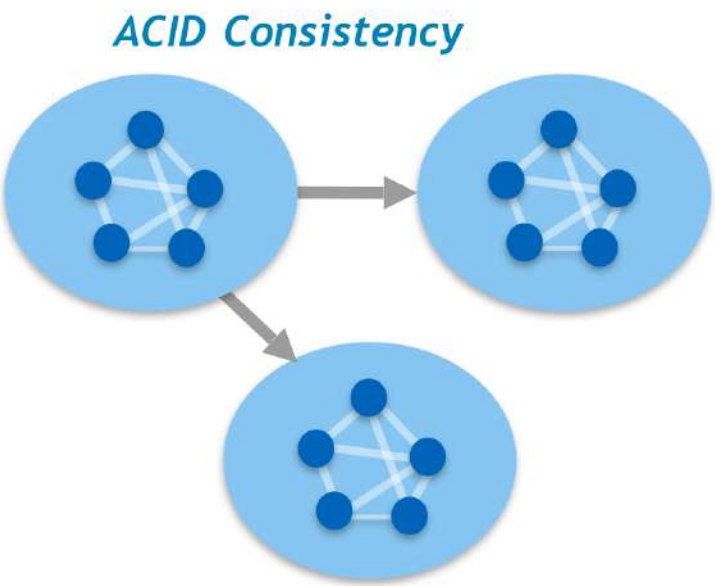
Neo4j Database: Index-free adjacency

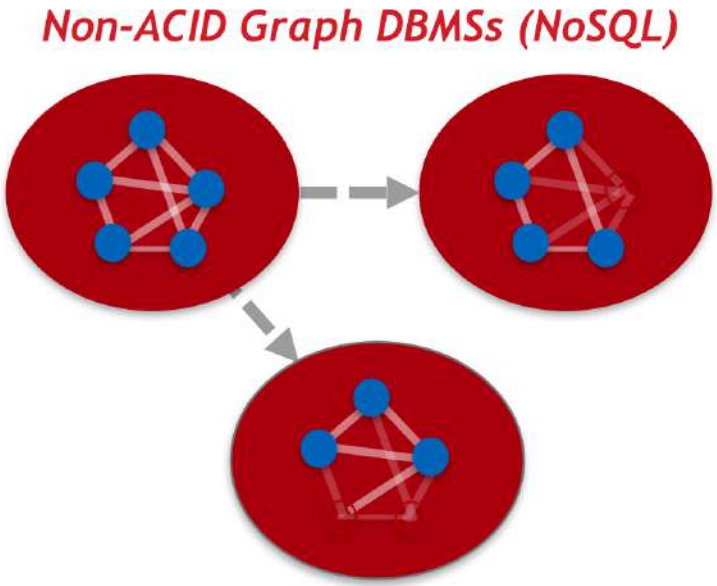
Nodes and relationships are stored on disk as a graph for fast navigational access using pointers.



Neo4j Database: ACID

Transactional consistency - all updates either succeed or fail.





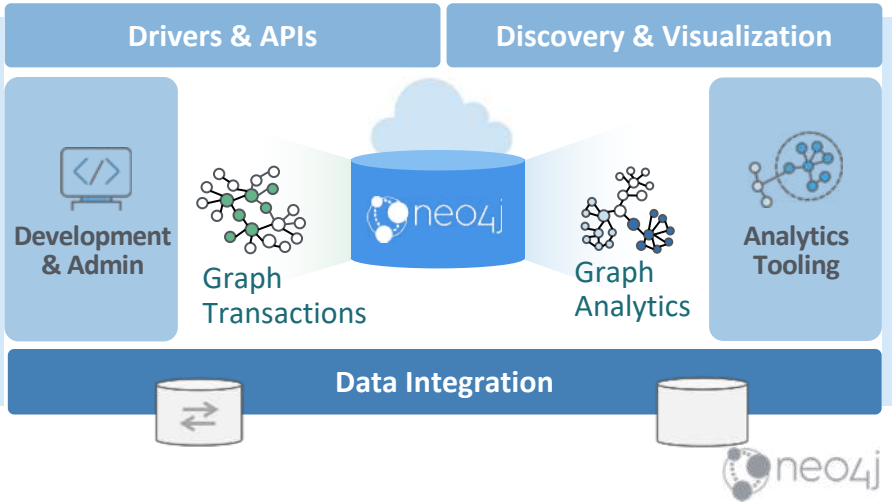
Native Graph Technology

Neo4j is an ***enterprise-grade native graph database and tools***

- Store, reveal and query data relationships
- Traverse and analyze highly connected data in real-time
- Add context and connect data to support emerging AI applications

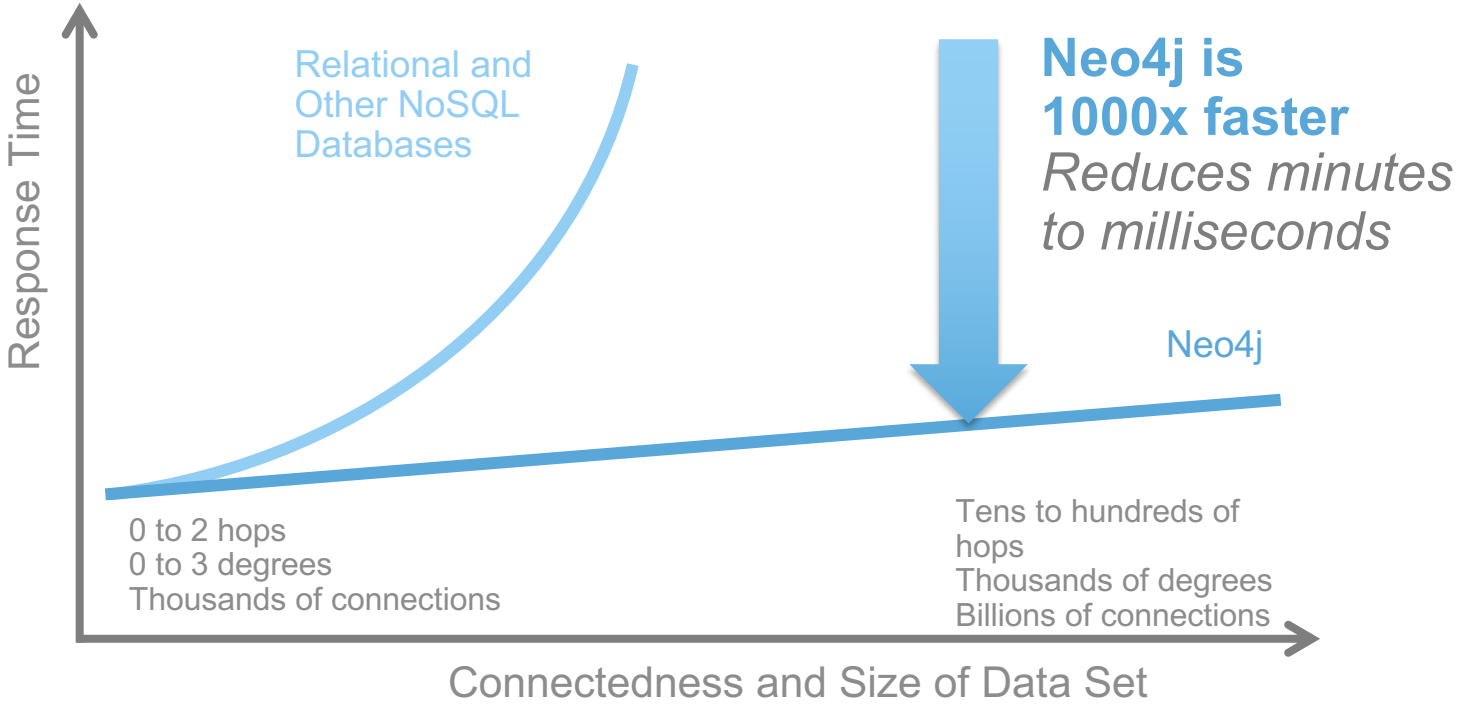
Neo4j Differentiators:

- Performance
- Visualization
- ACID Transactions
- Schema-free Agility
- Graph Data Science
- Global Scale
- Developer Productivity
- Deploy Anywhere

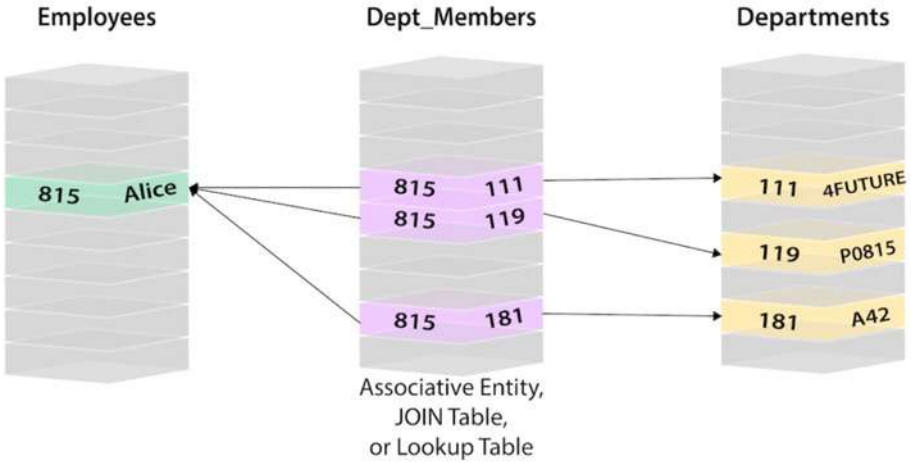


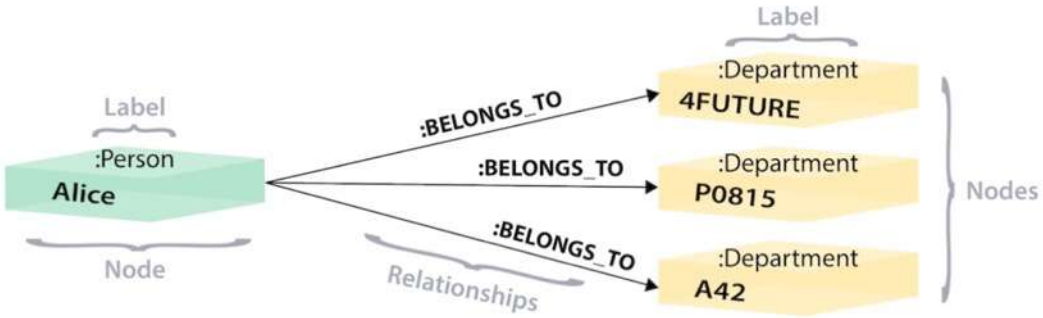
Real-Time Query Performance

Neo4j Versus Relational and Other NoSQL Databases



Relational to Graph






Real Time
Performance

Handling Large Graph Work Loads for Enterprises

Real-time promotion recommendations

- Record “Cyber Monday” sales
- About 35M daily transactions
- Each transaction is 3-22 hops
- Queries executed in 4ms or less
- Replaced IBM Websphere commerce

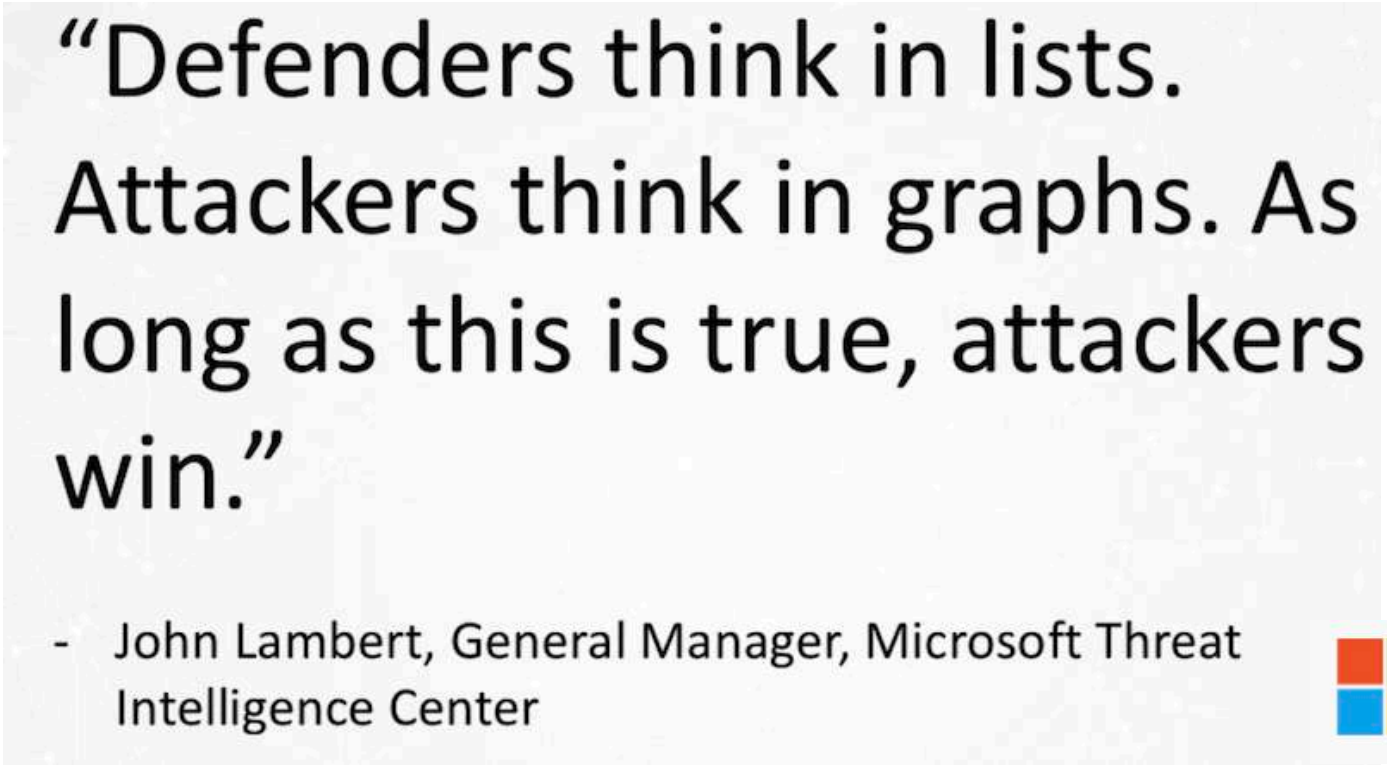
Marriott’s Real-time Pricing Engine

- 300M pricing operations per day
- 10x transaction throughput on half the hardware compared to Oracle
- Replaced Oracle database

Handling Package Routing in Real-Time

- Large postal service with over 500k employees
- Neo4j routes 7M+ packages daily at peak, with peaks of 5,000+ routing operations per second.

Pros use Neo4j for Cyber-security



https://neo4j.com/blog/bloodhound-how-graphs-changed-the-way-hackers-attack/

Productivity Gains with Cypher

The query asks: "Find all direct reports and how many people they manage, up to three levels down"

Example HR Query in SQL

```
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.pid AS directReportees, 0 AS count
  FROM person_reportee manager
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
  SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
SELECT reportee.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
  FROM(
  SELECT reportee.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages)
AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT L2Reportees.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
)
```

The Same Query using Cypher

```
MATCH (boss)-[:MANAGES*0..3]->(sub),
      (sub)-[:MANAGES*1..3]->(report)
WHERE boss.name = "John Doe"
RETURN sub.name AS Subordinate,
  count(report) AS Total
```

Project Impact

Less time writing queries

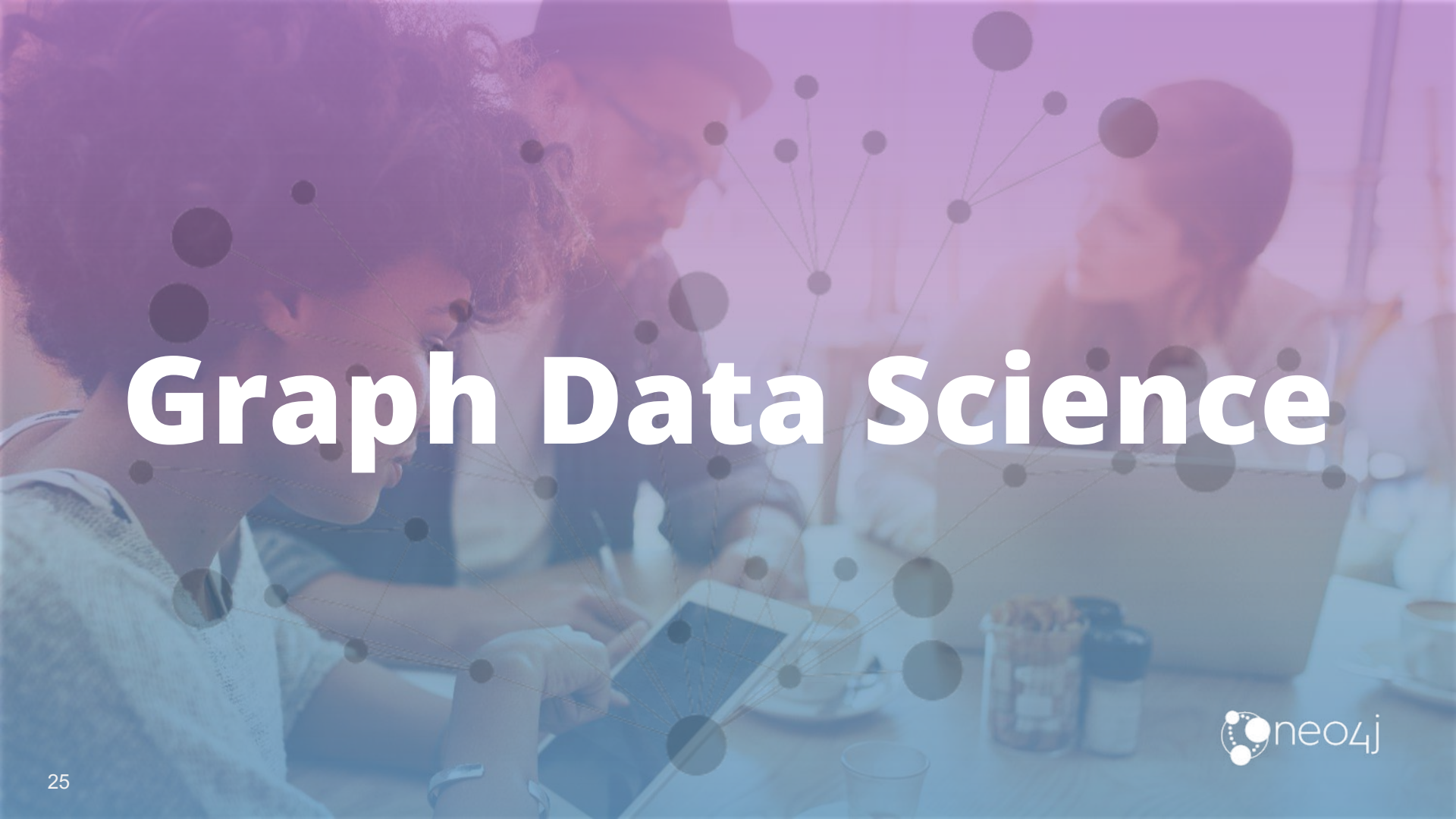
- More time understanding the answers
- Leaving time to ask the next question

Less time debugging queries:

- More time writing the next piece of code
- Improved quality of overall code base

Code that's easier to read:

- Faster ramp-up for new project members
- Improved maintainability & troubleshooting


Graph Data Science

Improving Analytics, ML & AI Across Industries

AstraZeneca Patient Journeys

- Early intervention project with 3 yrs of visits, tests & diagnosis with 10's of Bn of records
- Finding similarities in patient journeys
- Graph algorithms for identifying communities & best intervention points

Meredith Marketing to the Anonymous

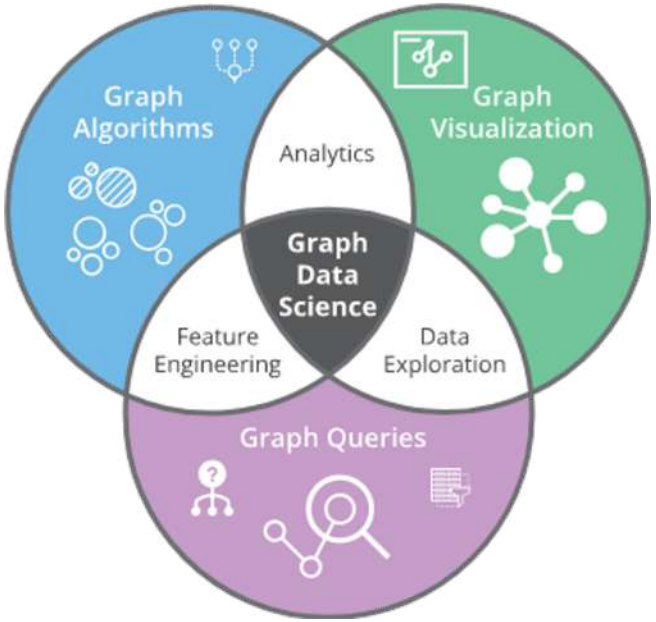
- Mostly anonymous users across devices and sites with ever changing cookies
- 4.4 TB: +14 Bn nodes +20Bn relationships
- +160 Mn rich, unique profiles created
- 612% Increase in visits per profile

Fraud Detection & Asset Recovery

Multiple Banks

- Majority of credit card fraud went undetected
- Millions of account with billions of transactions
- Graph analytics with queries & algorithms help find $10's of millions of fraud in 1st year

What is *Graph* data science?



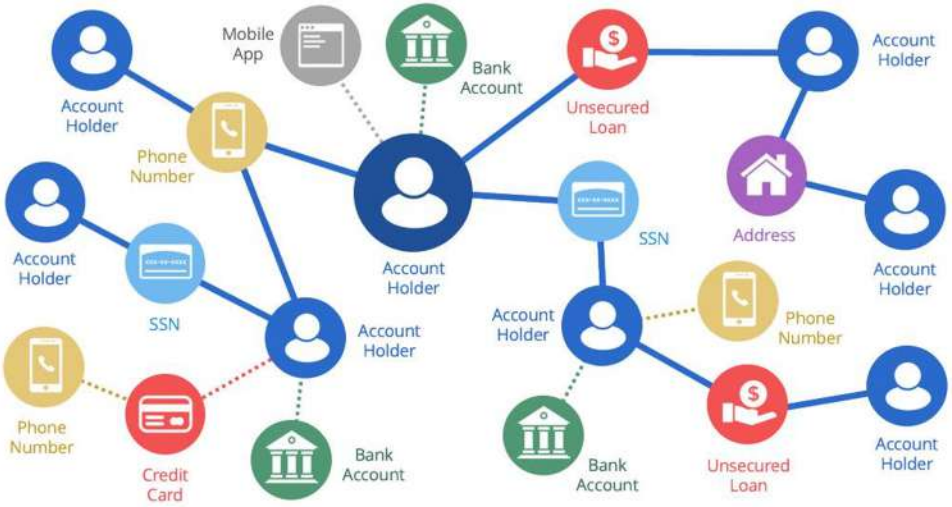
Graph Data Science is a science-driven approach to gain knowledge from the relationships and structures in data, typically to power predictions.

Data scientists use relationships to answer questions.

Knowledge Graph Queries

e.g. Detecting Financial Fraud

Improving existing pipelines to identify fraud via heuristics



Deceptively Simple Queries

How many flagged accounts are in the applicant's network **4+ hops out**?

How many **login / account variables in common**?

Add these metrics to your approval process

Difficult for RDBMS systems over 3 hops

So, When Do I Need Graph Algorithms?

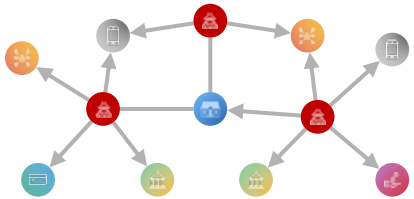
Query (e.g. Cypher/Python)

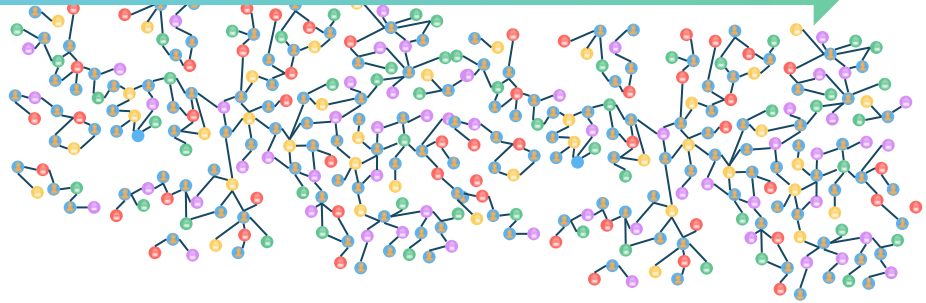
Real-time, local decisioning and pattern matching

Graph Algorithms

Global analysis and iterations

Local Patterns ⟵ ⟶ **Global Computation**





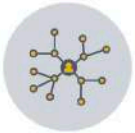
You know what you're looking for and making a decision

You're learning the overall structure of a network, updating data, and predicting

The Neo4j Graph Data Science Library

Pathfinding & Search

- Deep path analytics
- Optimal routing



Centrality / Importance

- Identifies node importance
- Influencer & Risk Identification

Community Detection

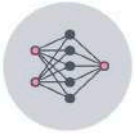
- Detects group clustering
- Partition options

Link Prediction

- Estimates likelihood of
- Estimate missing information

Similarity

- Evaluates how alike nodes are
- Construct graphs from data

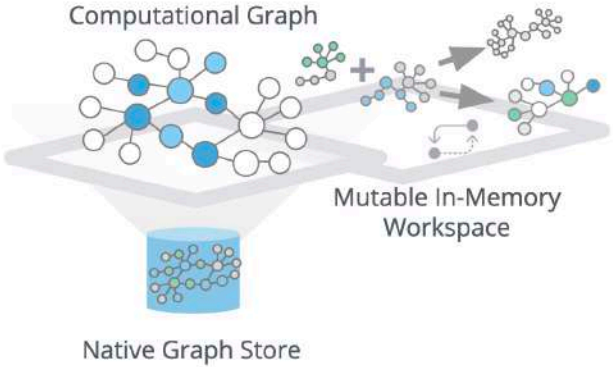


Graph Embeddings

- Learn your graph topology
- Use for dimensionality reduction

50+ Robust Algorithms

Flexible Analytics Workspace



Graph Algorithms
e.g. Detecting Financial Fraud

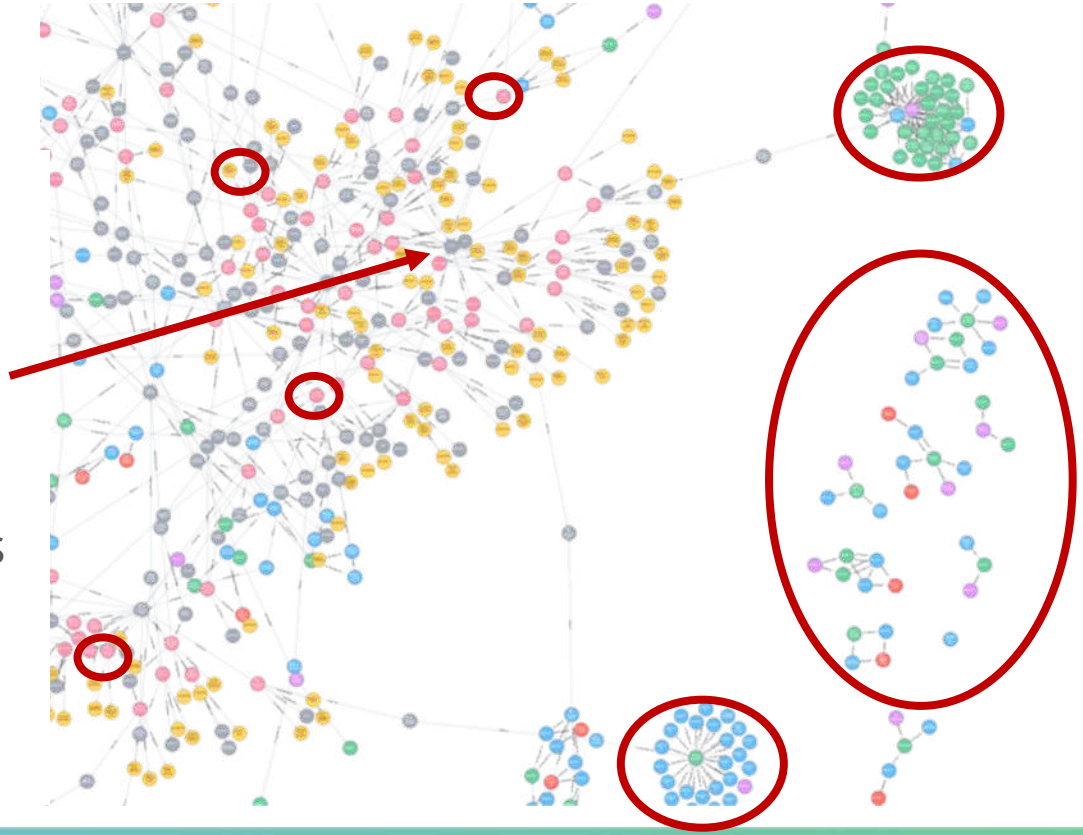
Graph algorithms enable reasoning about **network structure**

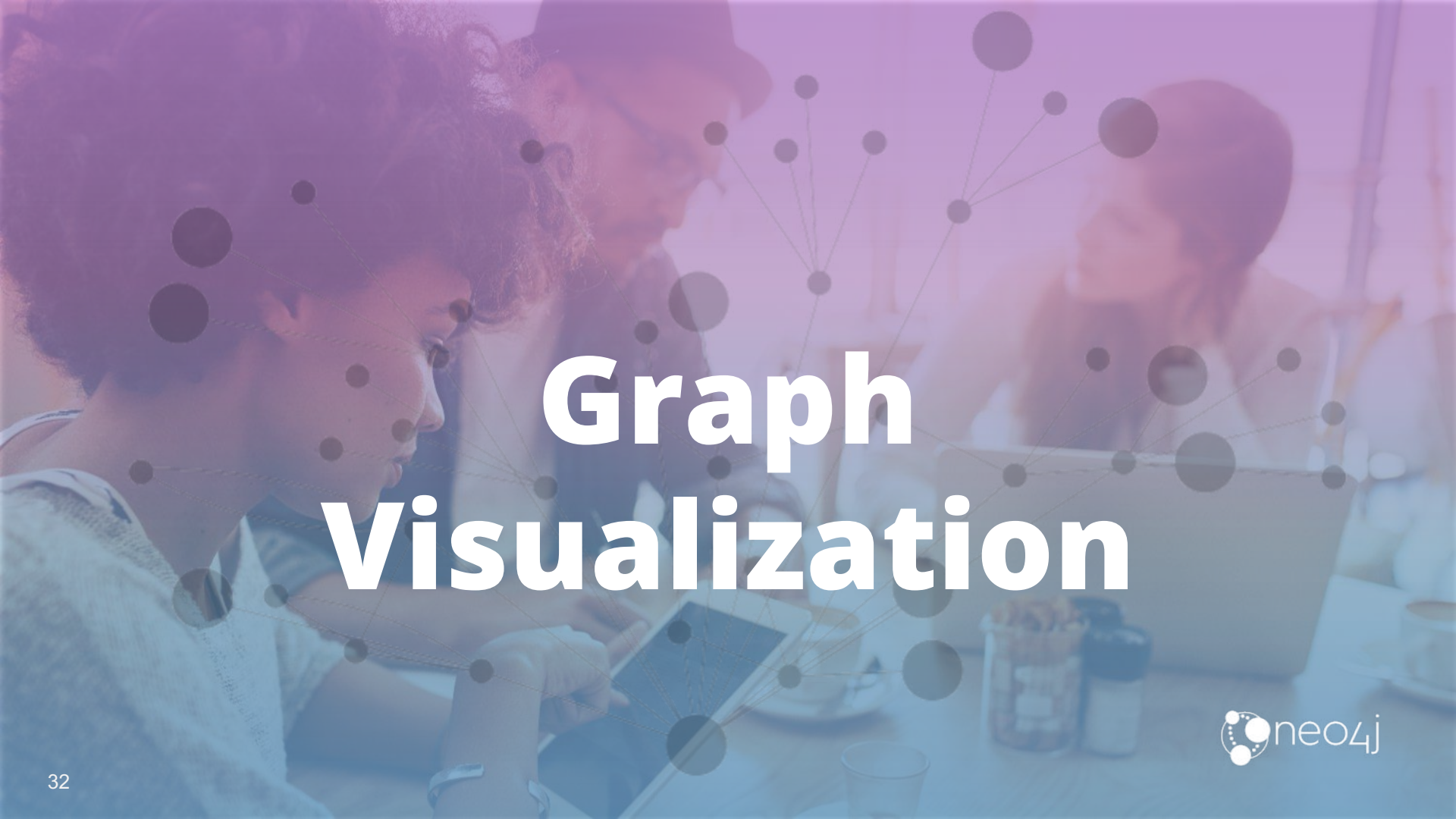
Connected components identify disjointed group sharing identifiers

PageRank to measure influence and transaction volumes

Louvain to identify communities that frequently interact

Jaccard to measure account similarity




Graph
Visualization

Neo4j Bloom's Intuitive User Interface

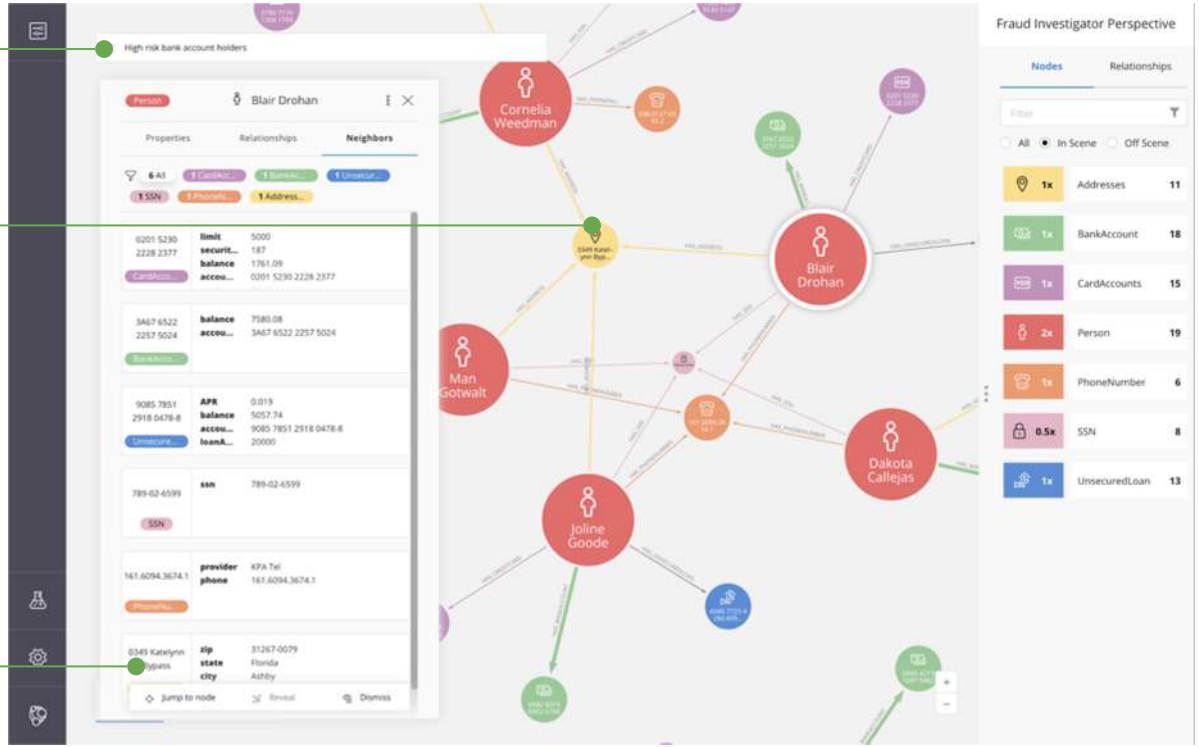
Search with type-ahead suggestions

Flexible Color, Size and Icon schemes

Visualize, Explore and Discover

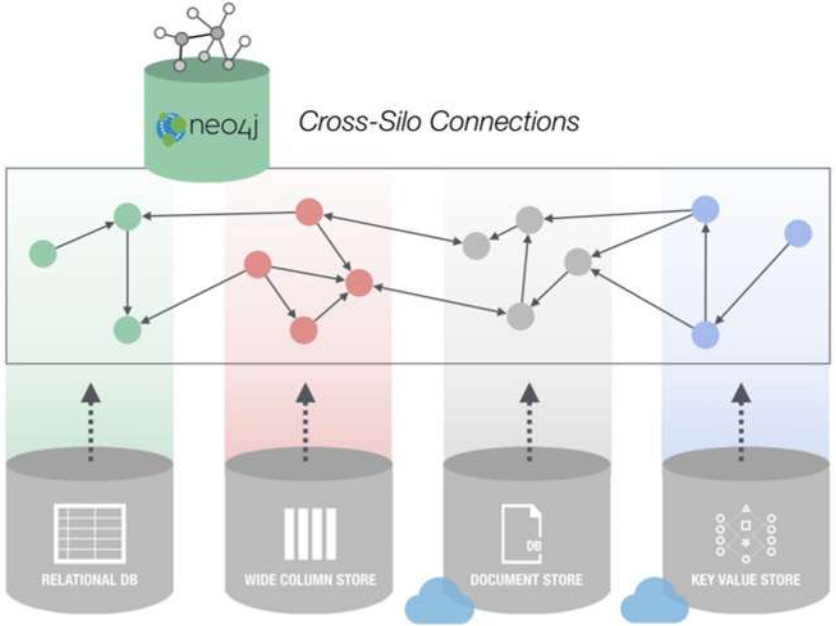
Pan, Zoom and Select

Property Browser and editor

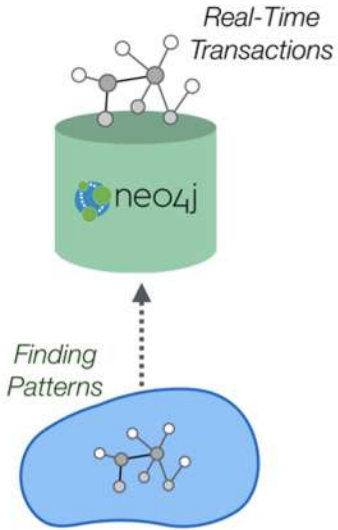


How Neo4j Fits — Common Architecture Patterns

From Tabular Data
To Connected Data



From Disparate Silos
To Cross-Silo Connections



From Data Lake Analytics
to Real-Time Operations

neo4j

Neo4j Ecosystem

Native Graph Technology for Applications & Analytics

Applications

Business Users

Drivers & APIs

Discovery & Visualization

Developers

Admins

Development & Admin

Graph Transactions

neo4j

APOC/Stored Procedures

Graph Analytics

Analytics Tooling

Data Analysts

Data Scientists

BIConnector

Data Integration

kafka

Enterprise Data Hub

NEOSEMANTX

neo4j

Neo4j Drivers

Languages

- Java, JavaScript, .NET, Python, Go, R, Ruby, PHP, Erlang, Perl

Driver modes

- Simple
- Asynchronous
- Reactive (back-pressure and flow control)

Transaction routing

- Route request to appropriate server based on server load and if read or write operation

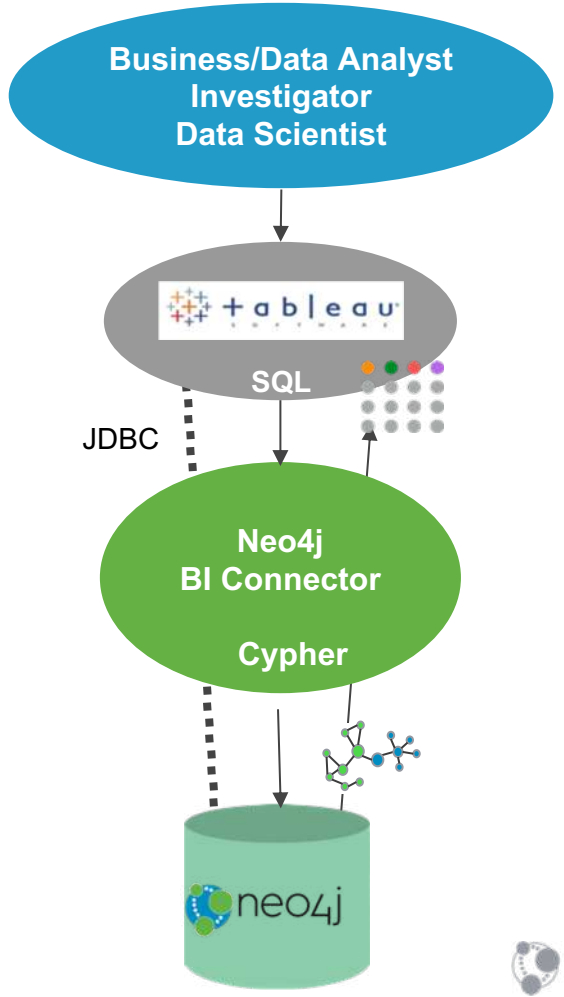
Uses

- Cypher based graph queries
- Coding Environments
  - Juypter, Colab, RSuite
- Engine API custom procedures, functions
  - Traversals, injections, etc.
- Extension to Graph Data Science Library (e.g. Pregel)

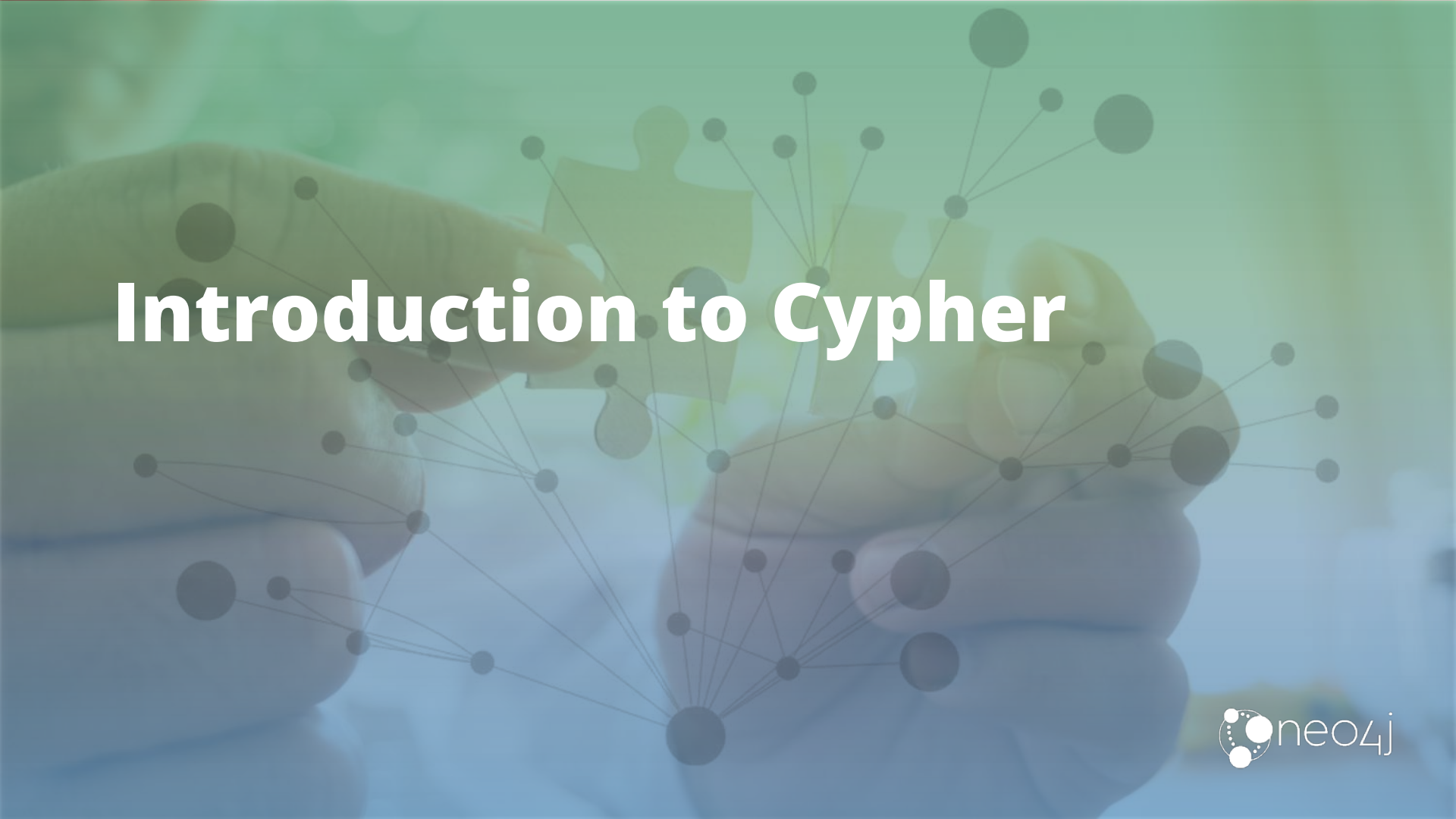
Neo4j BI Connector

The most popular BI tools can now talk live to the world's most popular graph database

- Best live, seamless integration of graph data with your favorite BI tools
  - Familiar UI for end users
  - No development effort for IT
- Democratizes access to Neo4j data
- Free to adopt by BI teams of Enterprise Edition customers

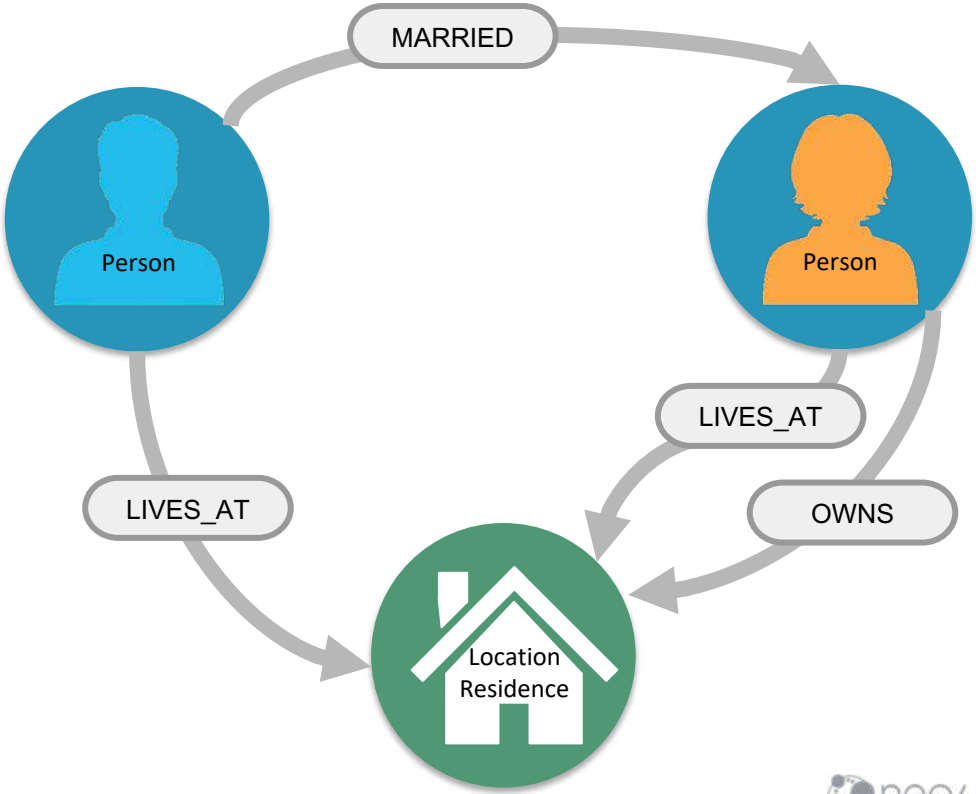


Introduction to Cypher

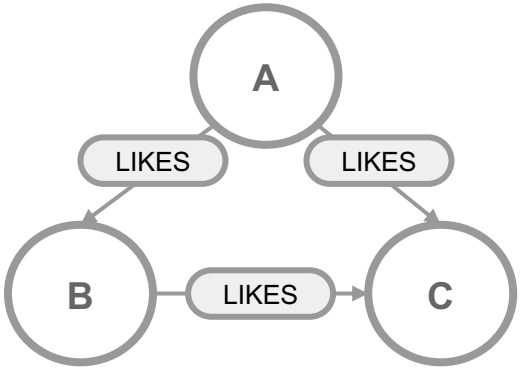


What is Cypher?

- 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

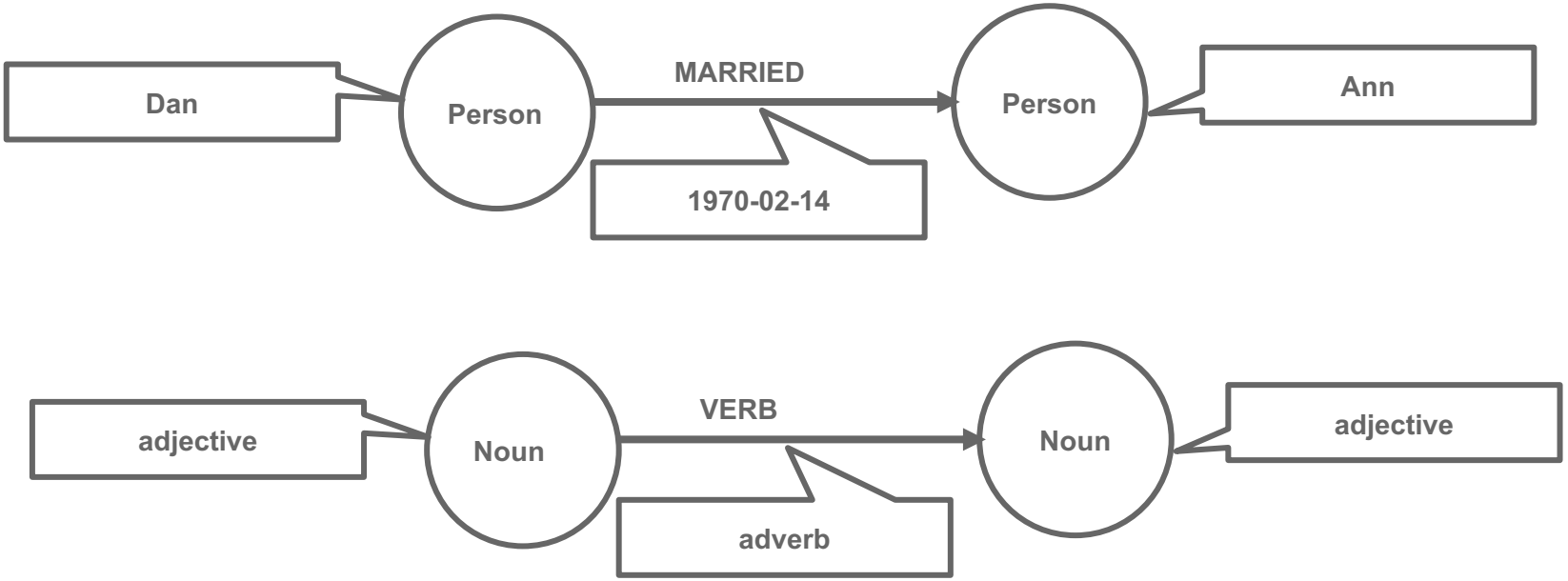


Cypher is ASCII Art



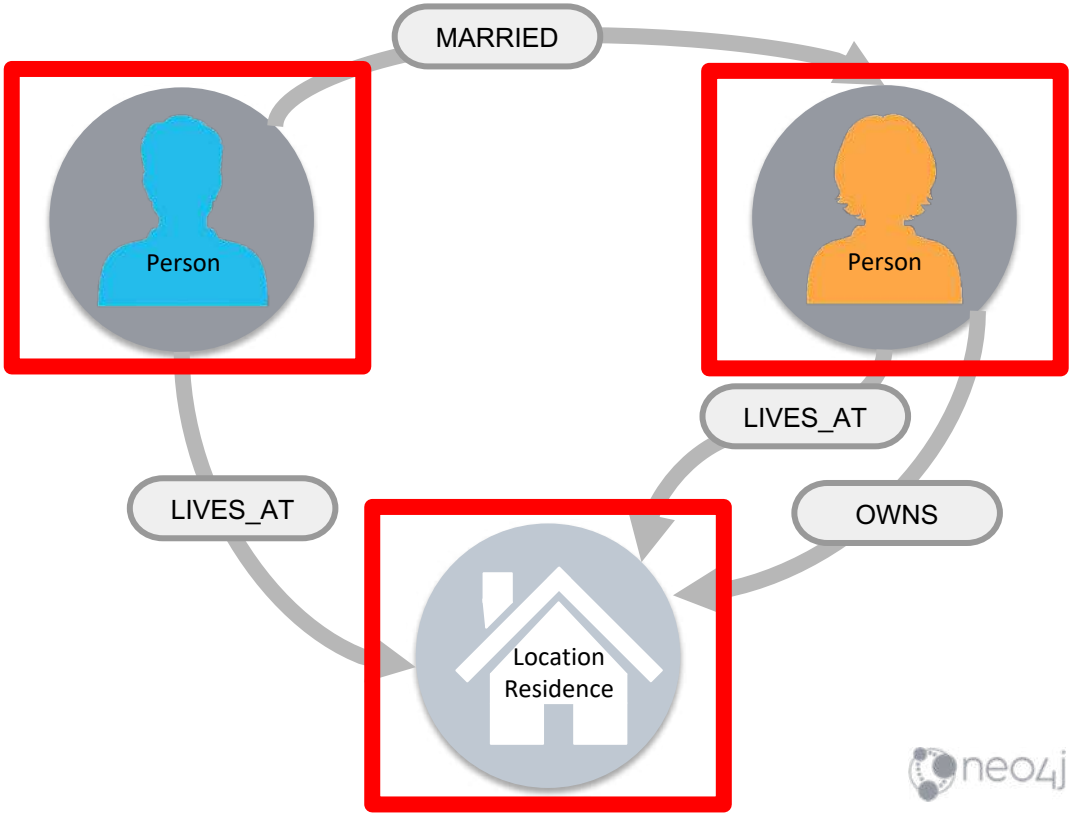

```
(A)-[:LIKES]->(B),(A)-[:LIKES]->(C),(B)-[:LIKES]->(C)

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

Cypher is readable



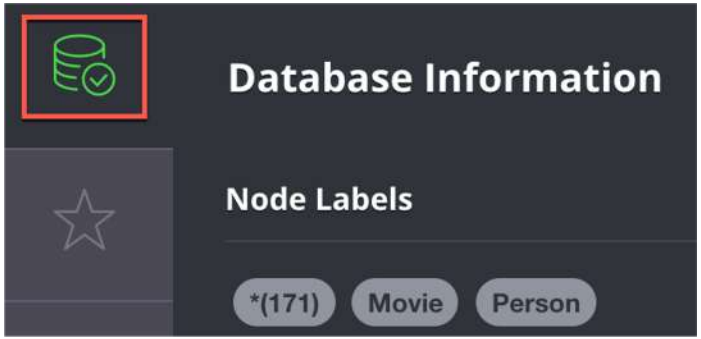
Nodes

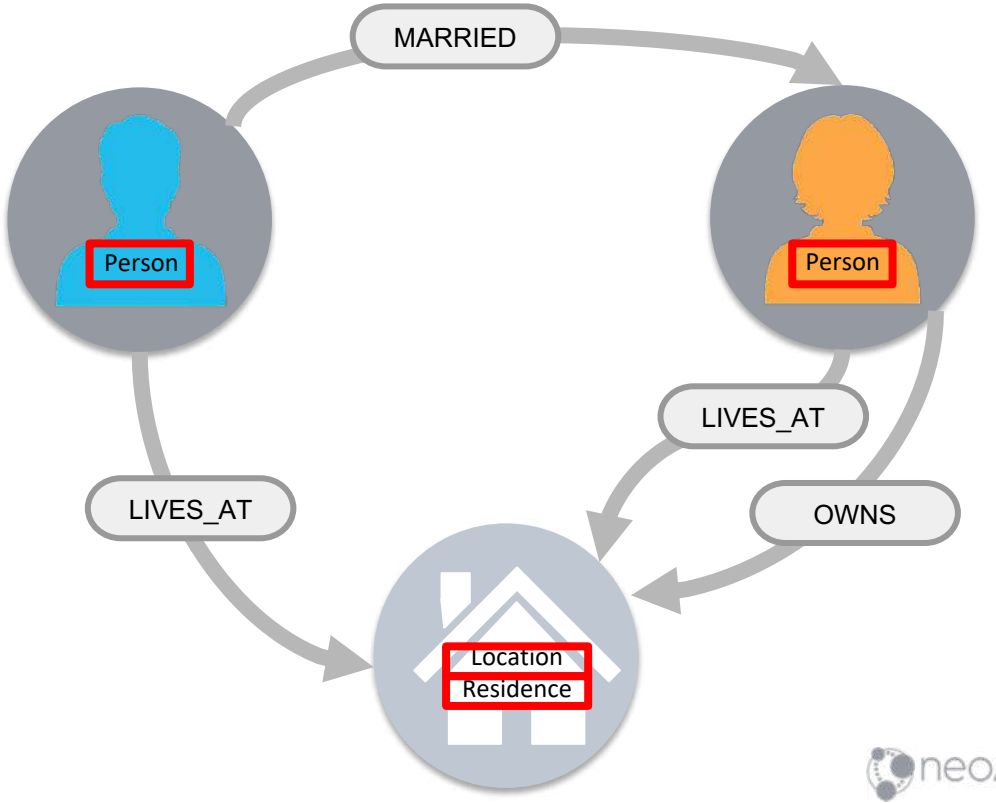
```
()
(p)
(l)
(n)
```


MARRIED
Person
Person
LIVES_AT
LIVES_AT
OWNS
Location
Residence

Labels

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

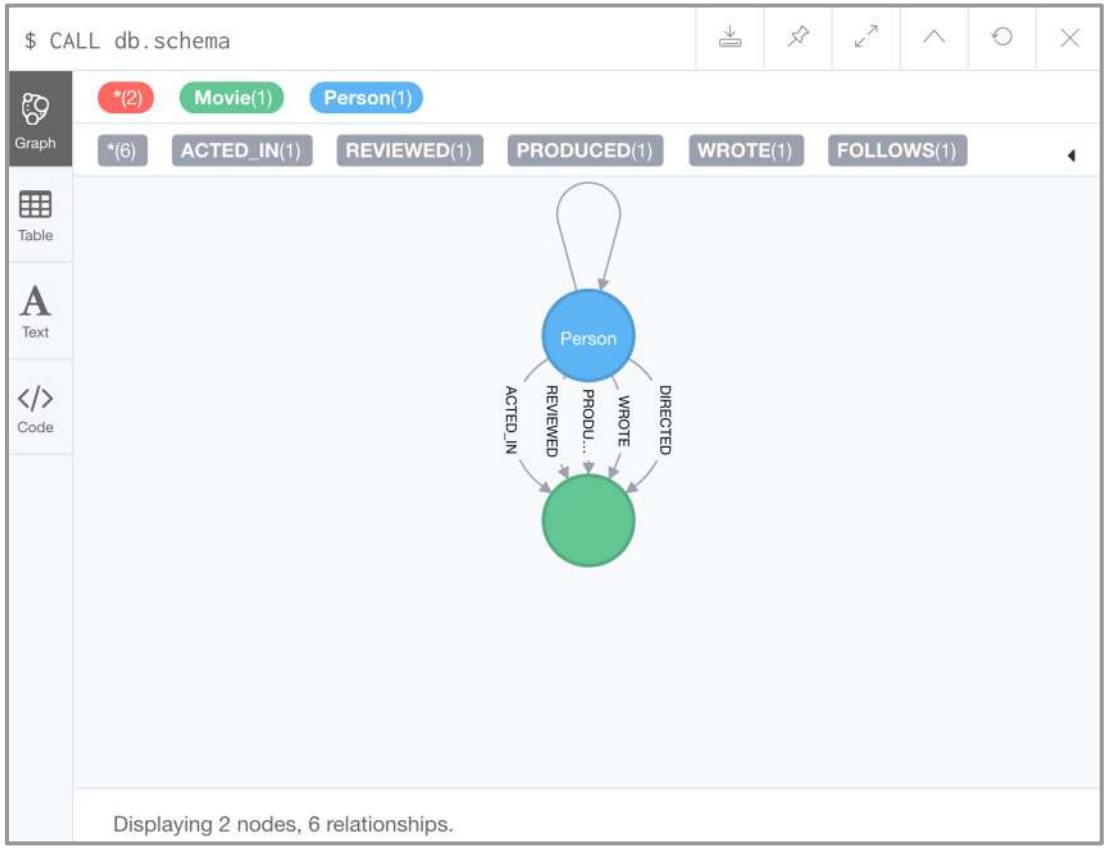




Comments in Cypher

```
()                  // 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

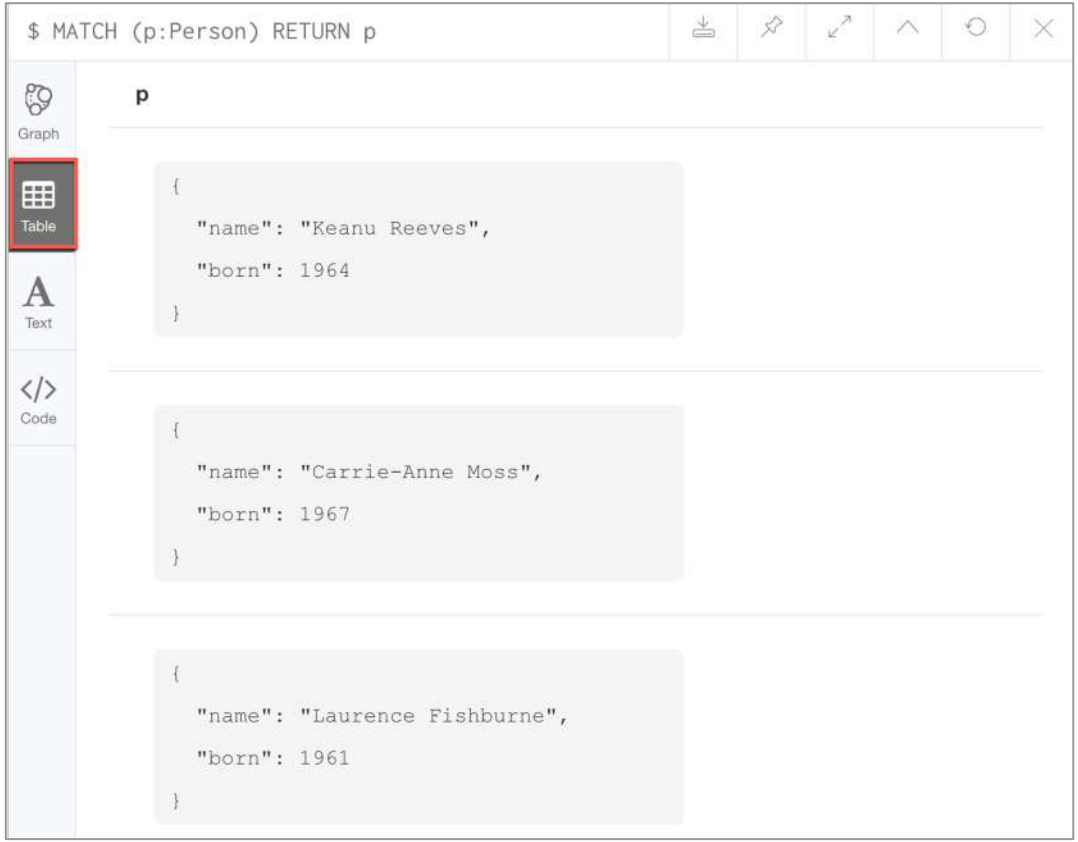


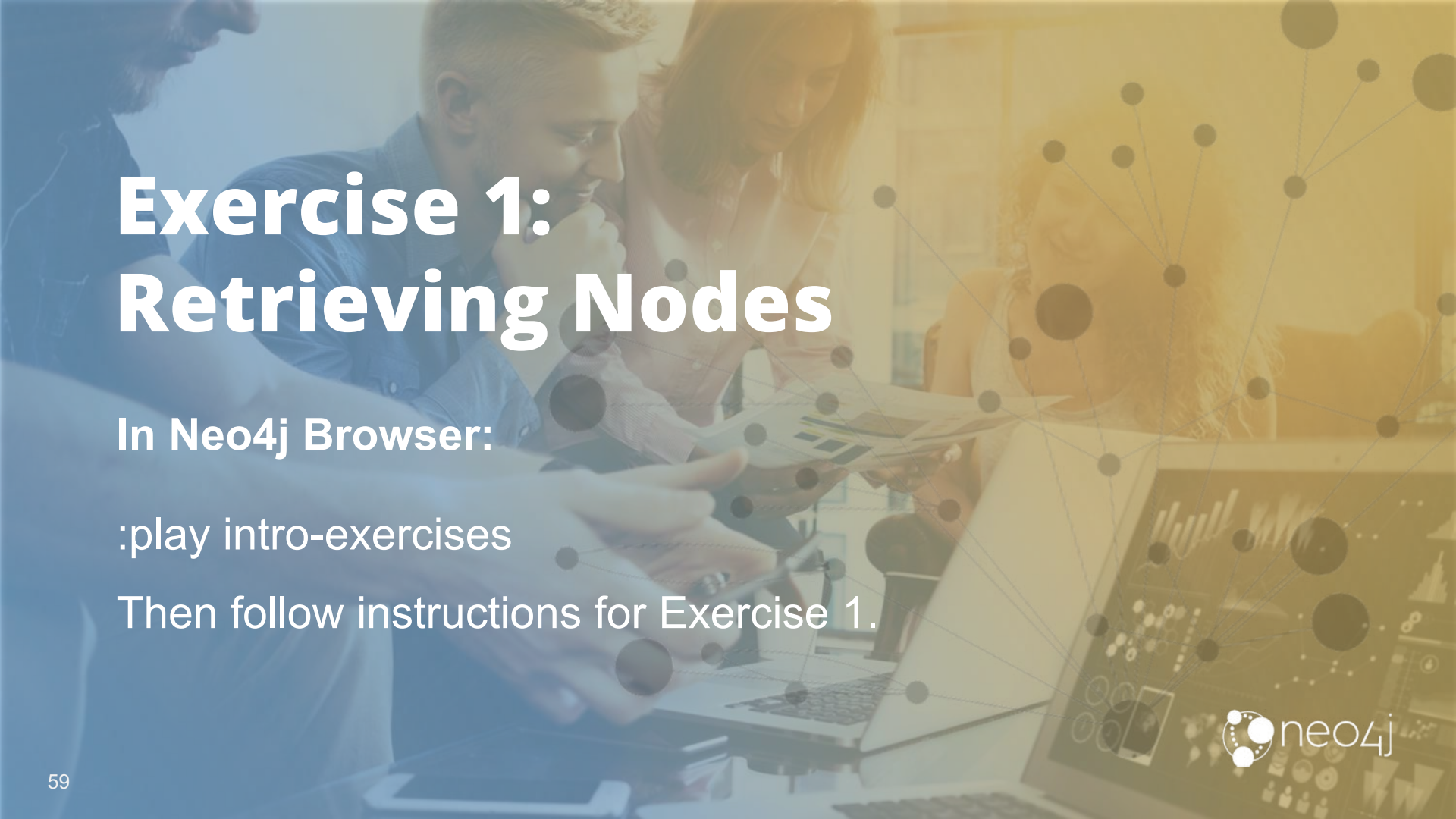
Using MATCH to retrieve nodes

```
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





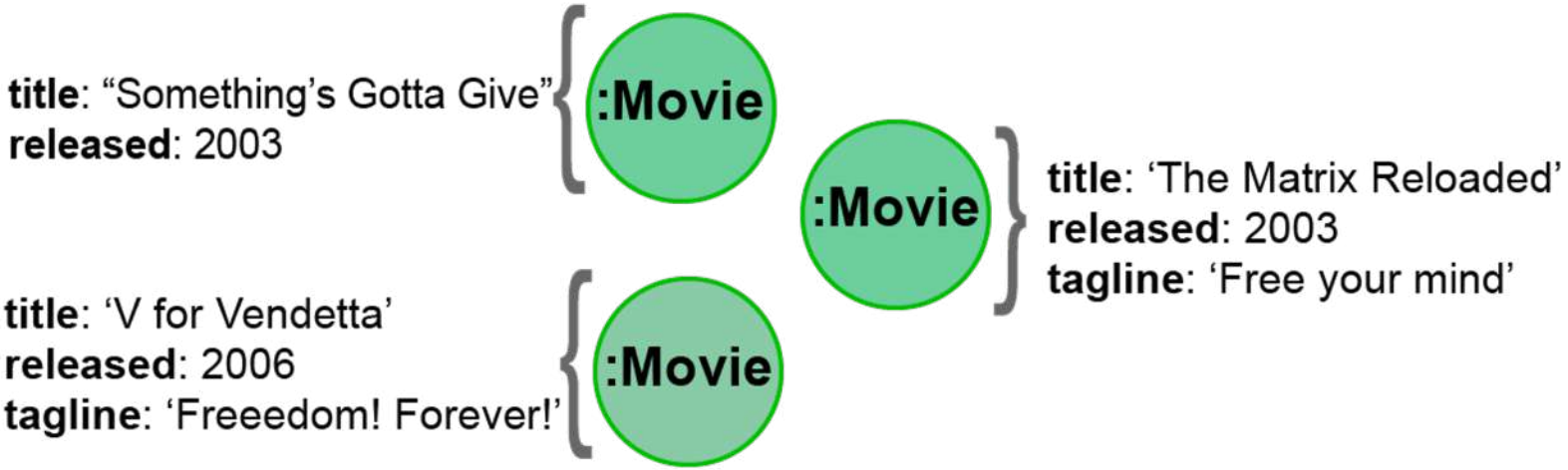
Exercise 1: Retrieving Nodes

In Neo4j Browser:

:play intro-exercises

Then follow instructions for Exercise 1.

Properties



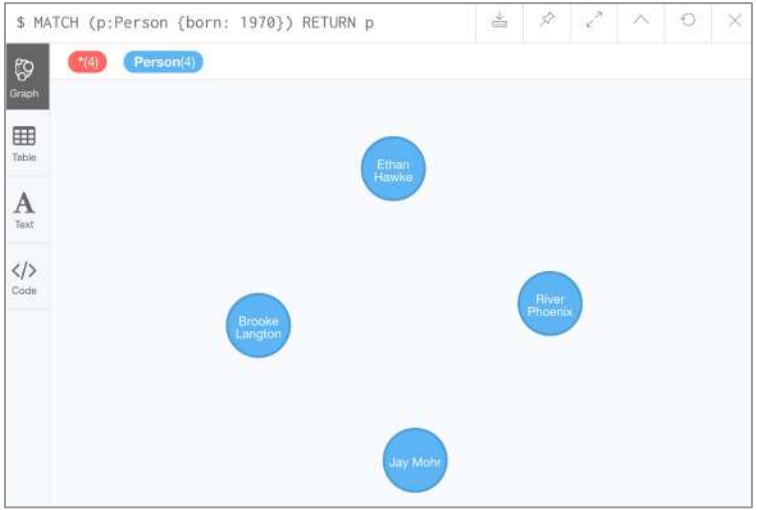
Examining property keys

```
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
```

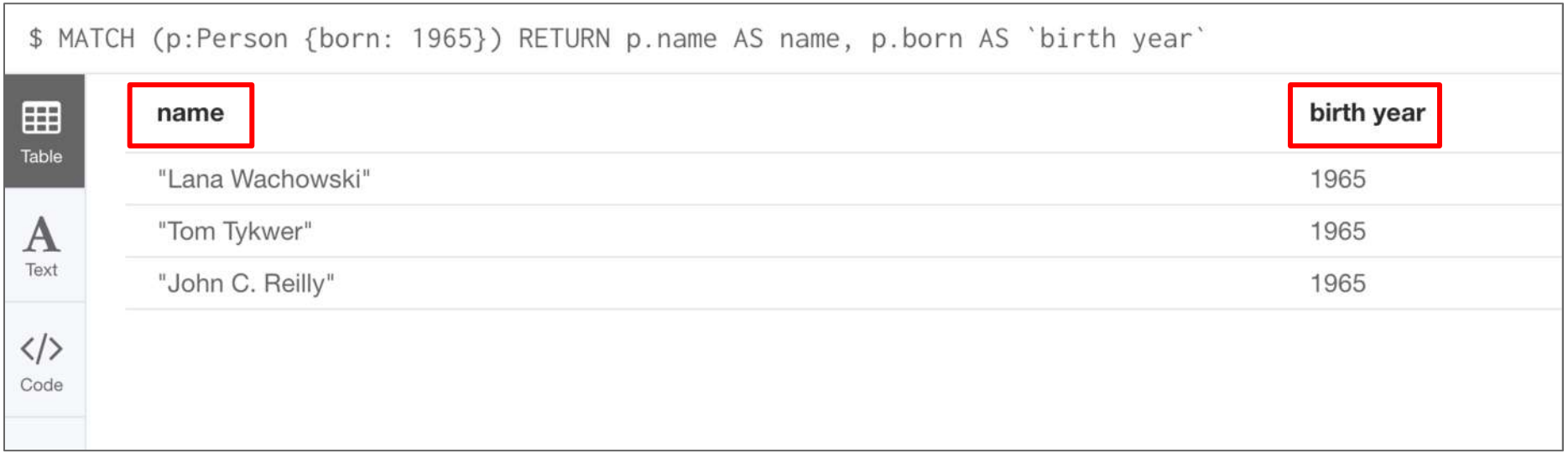
Returning property values

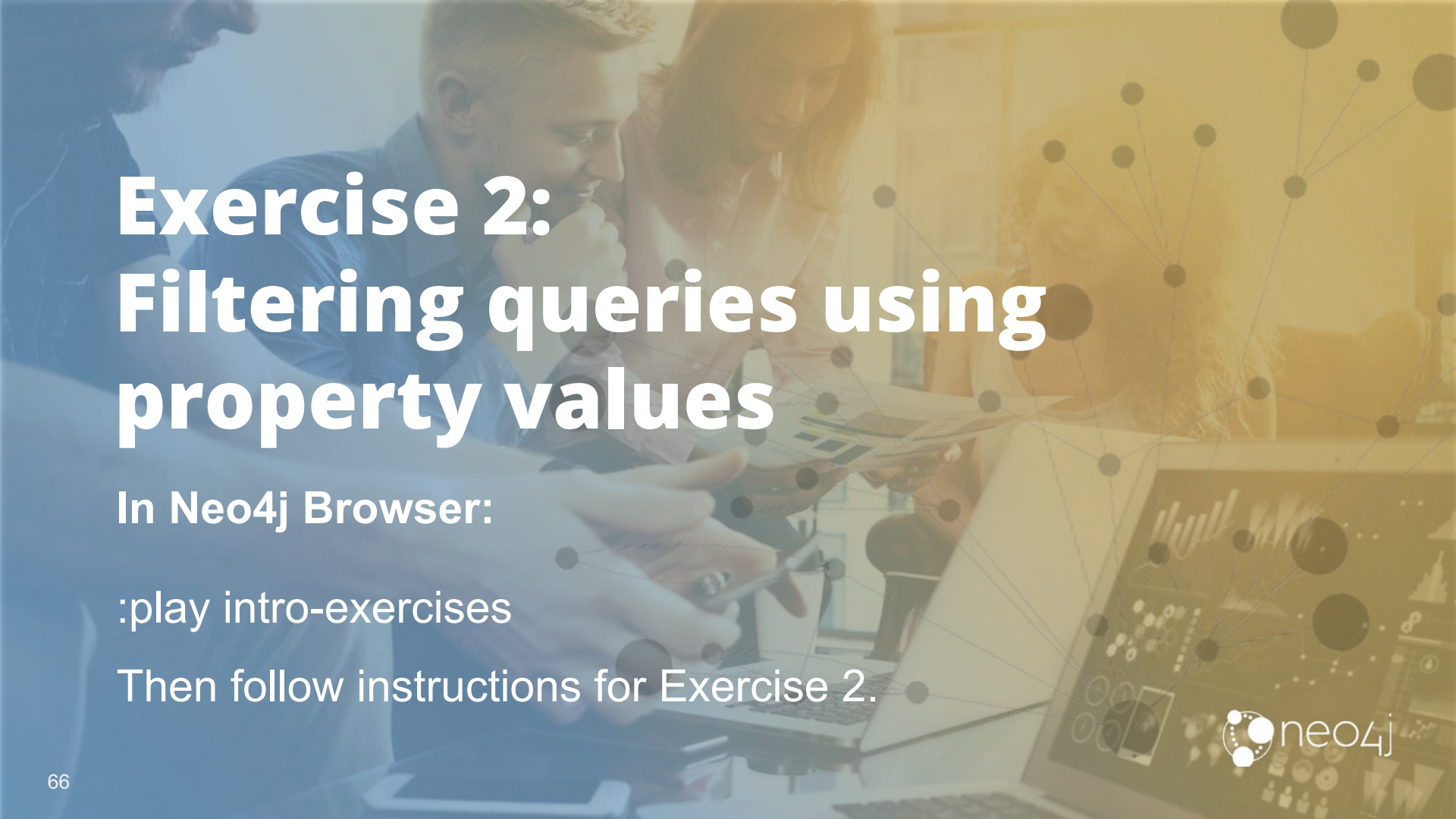
Find all people born in *1965* and return their names:

```
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`
```





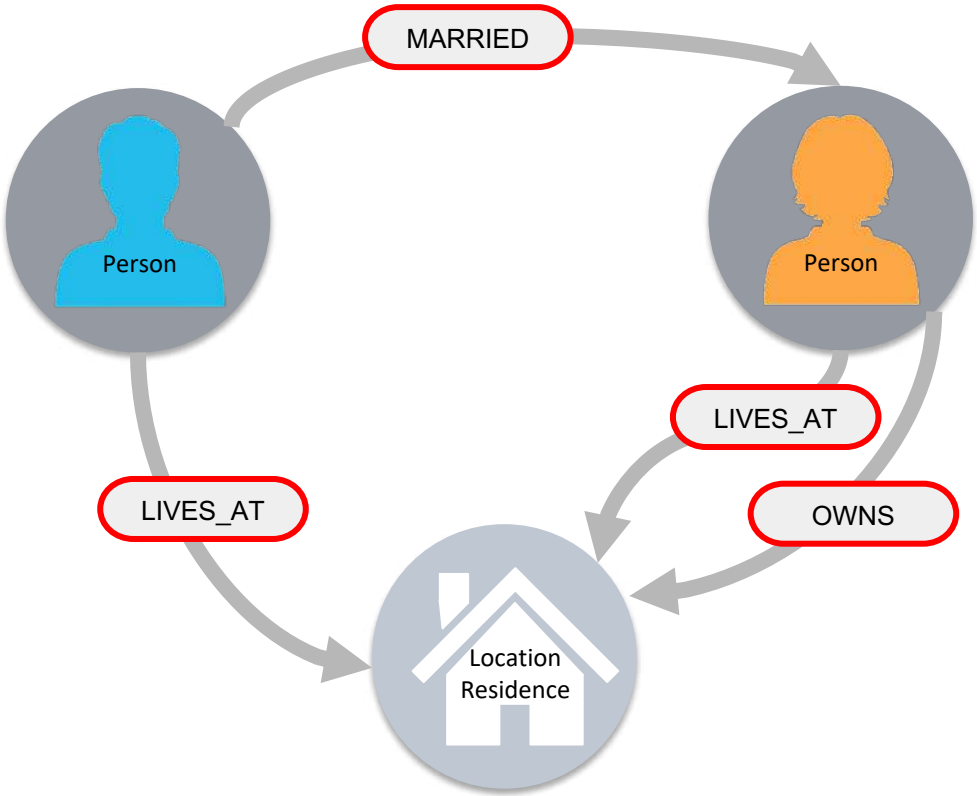
Exercise 2: Filtering queries using property values

In Neo4j Browser:

:play intro-exercises

Then follow instructions for Exercise 2.

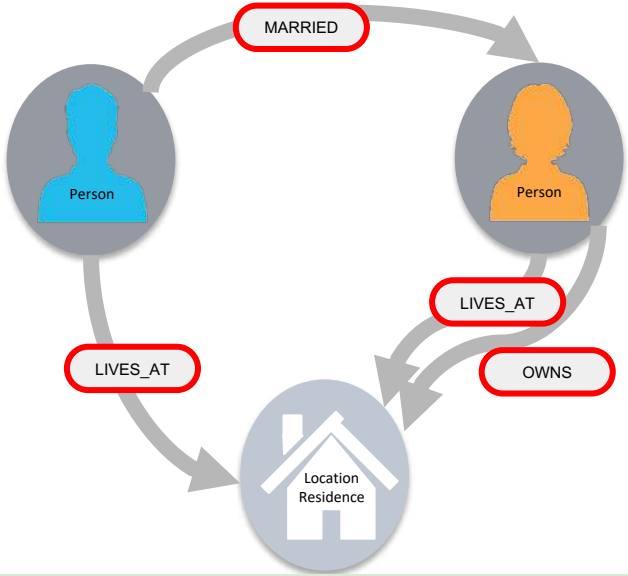
Relationships



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

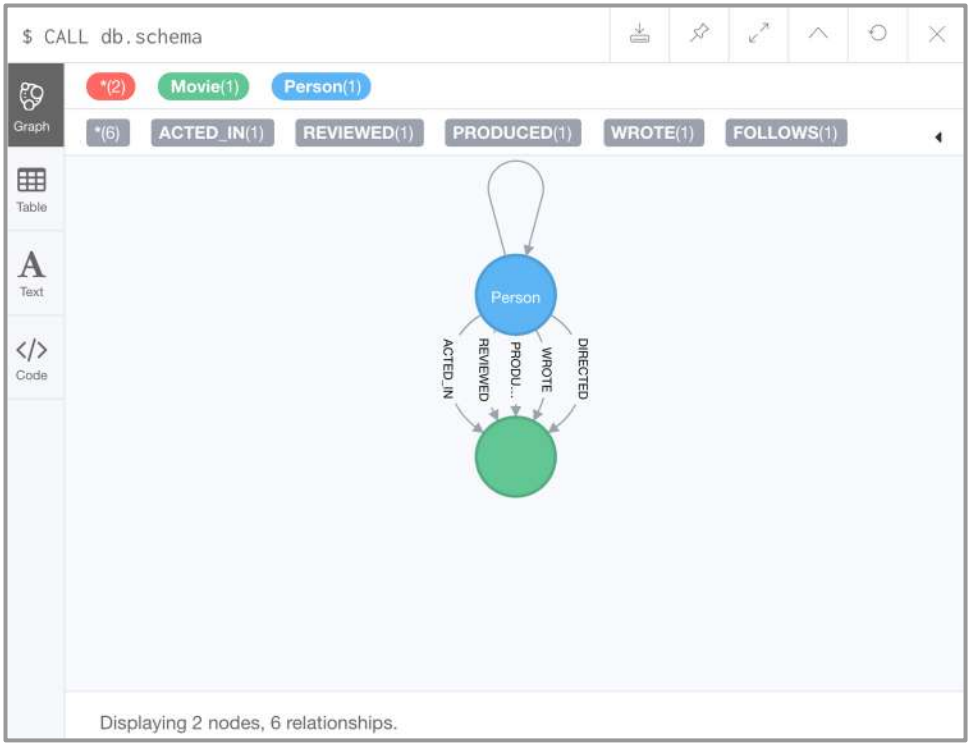



```
MATCH (p:Person)-[:LIVES_AT]->(h:Residence)
RETURN p.name, h.address
```

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

Examining relationships

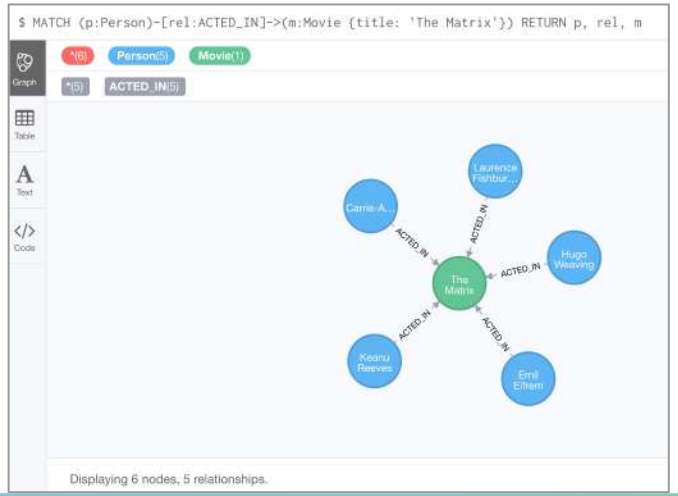
```
CALL db.schema.visualization()
```



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

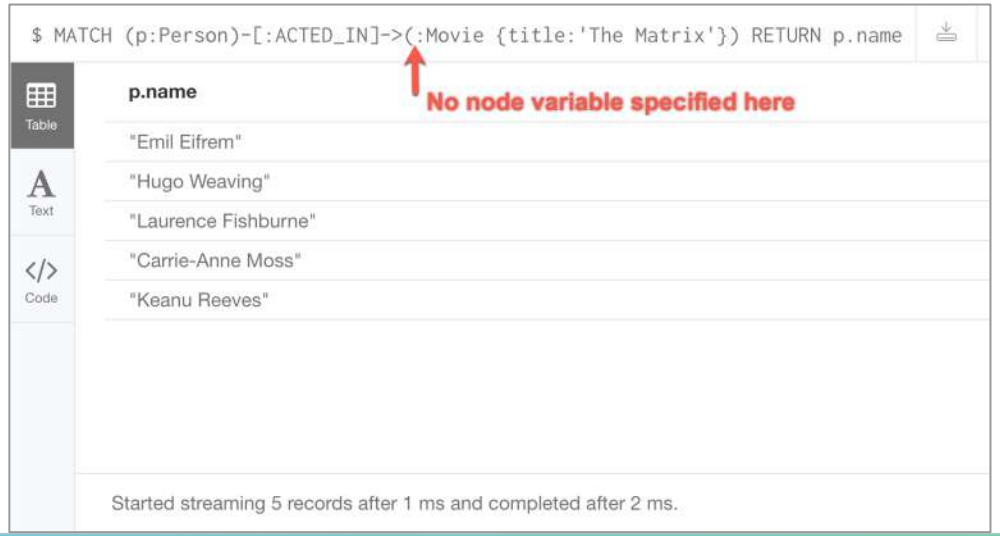
Find all movies that *Tom Hanks* acted in or directed and return the title of the move:

```
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:

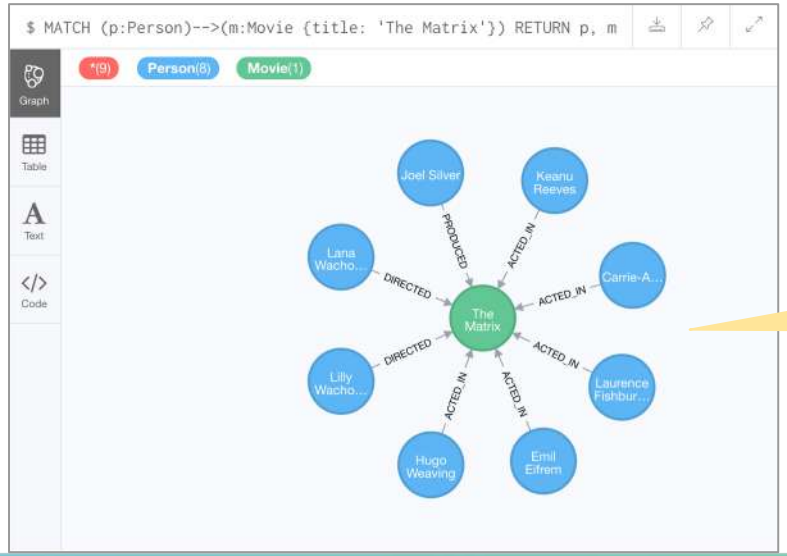
```
MATCH (p:Person)-[:ACTED_IN]->(:Movie {title: 'The Matrix'})
RETURN p.name
```



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
```

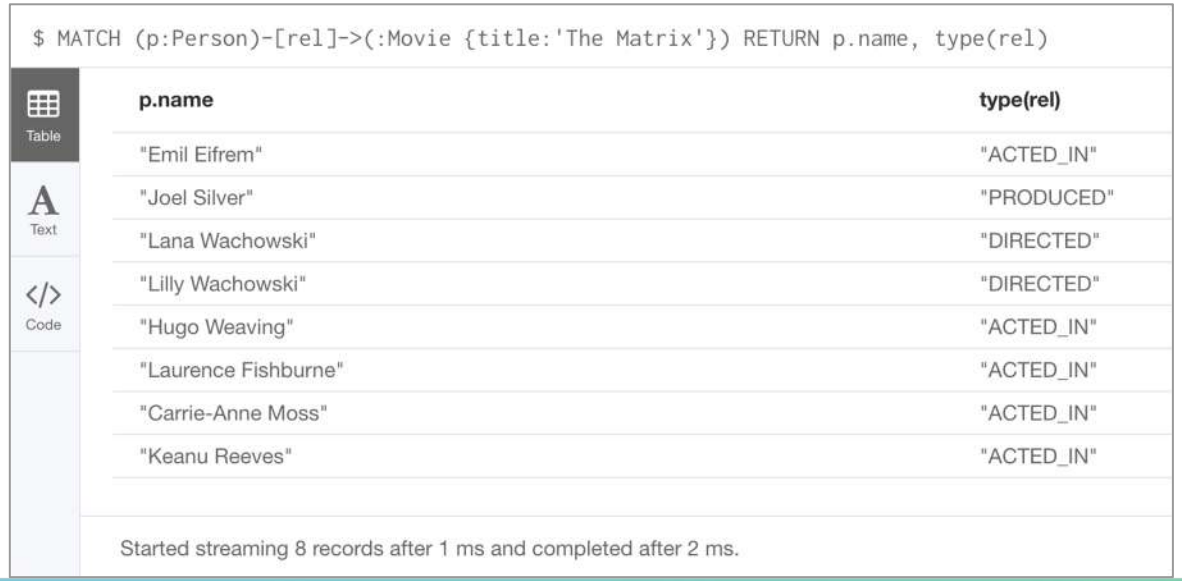


Connect result nodes enabled in Neo4j Browser

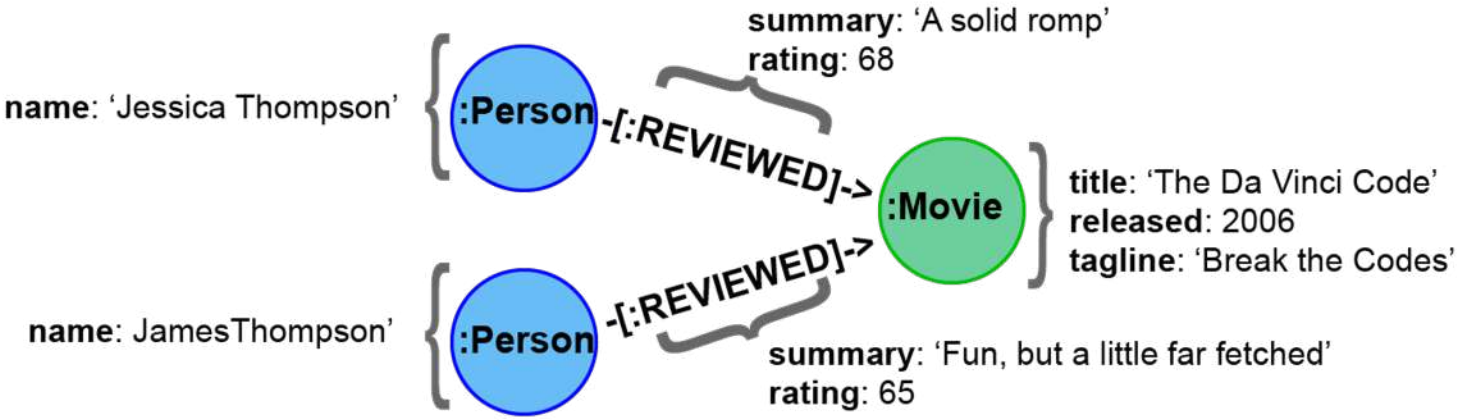
Retrieving relationship types

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)
```



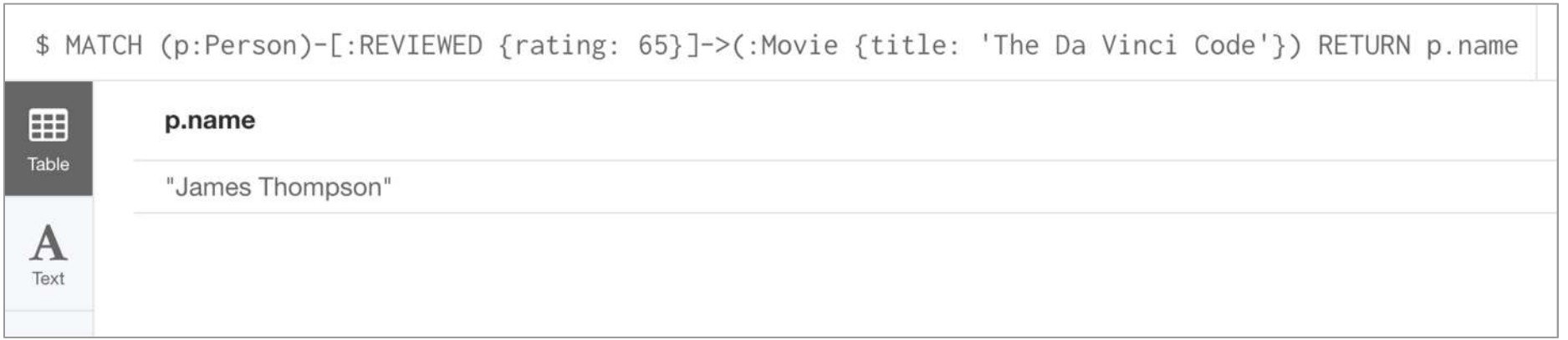
Retrieving properties for a relationship - 1



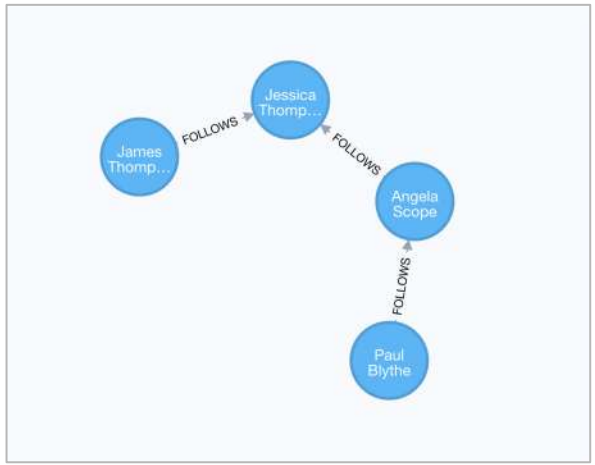
Retrieving properties for a relationship - 2

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

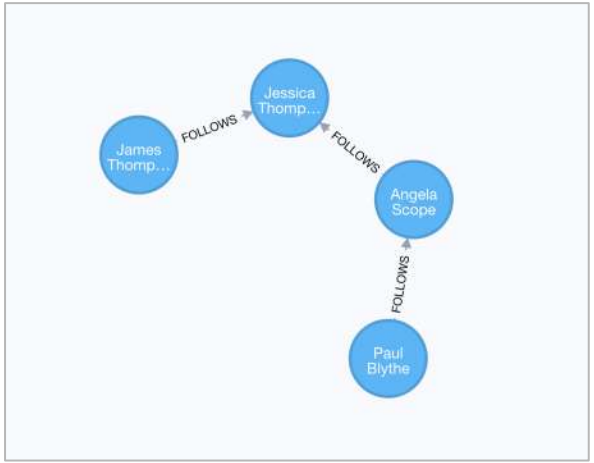


Find all people who follow *Angela Scope*, returning the nodes:

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

neo4j

Using patterns for queries - 2

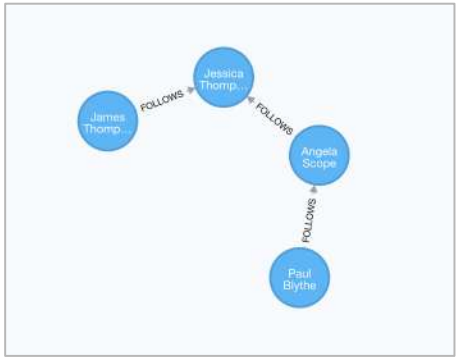


Find all people who *Angela Scope* follows, returning the nodes:

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

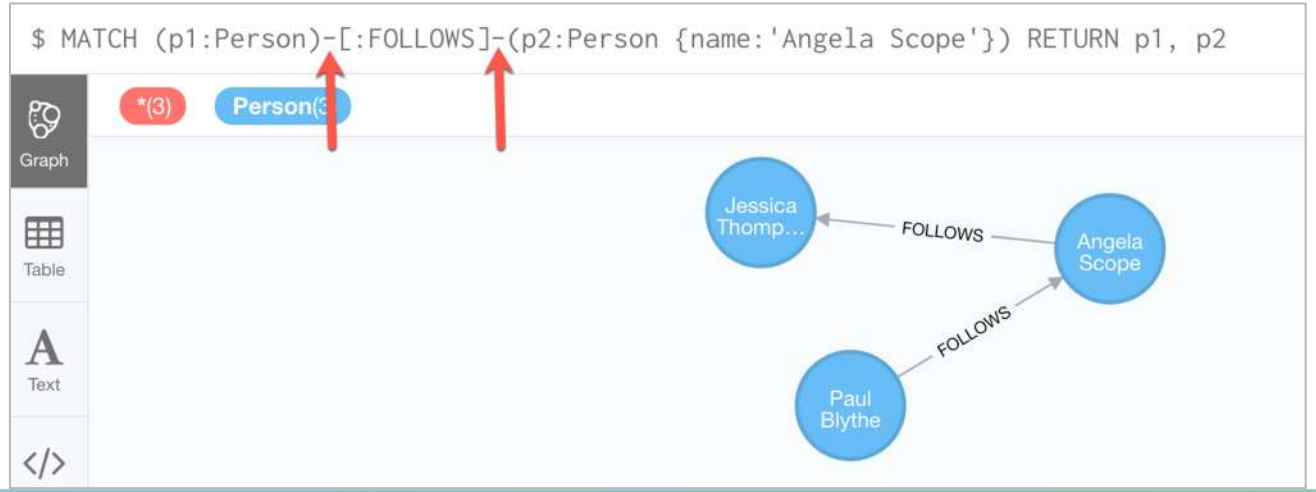
neo4j

Querying by any direction of the relationship



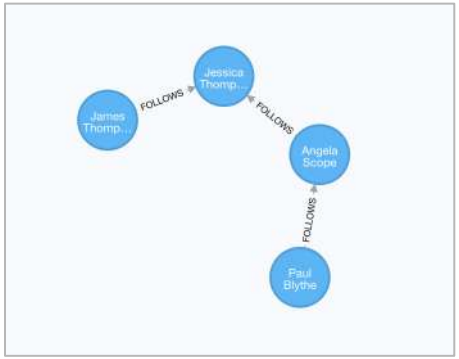
Find all people who follow *Angela Scope* or who *Angela Scope* follows, returning the nodes:

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



neo4j

Traversing relationships - 1

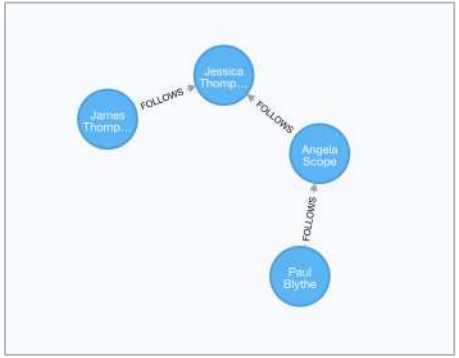


Find all people who follow anybody who follows *Jessica Thompson* returning the people as nodes:

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

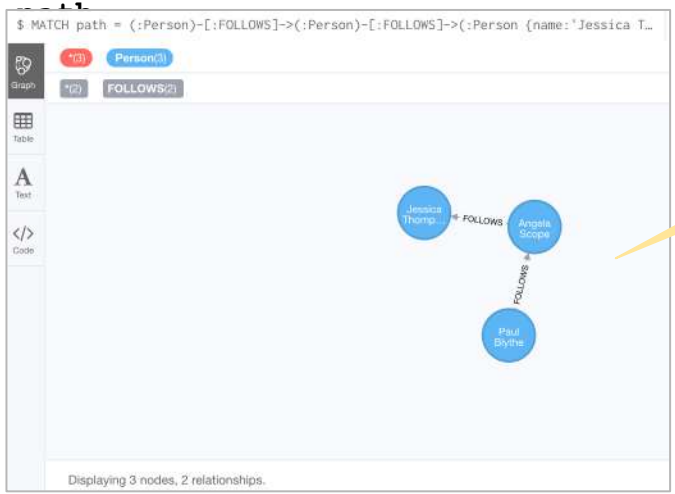
neo4j

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
```



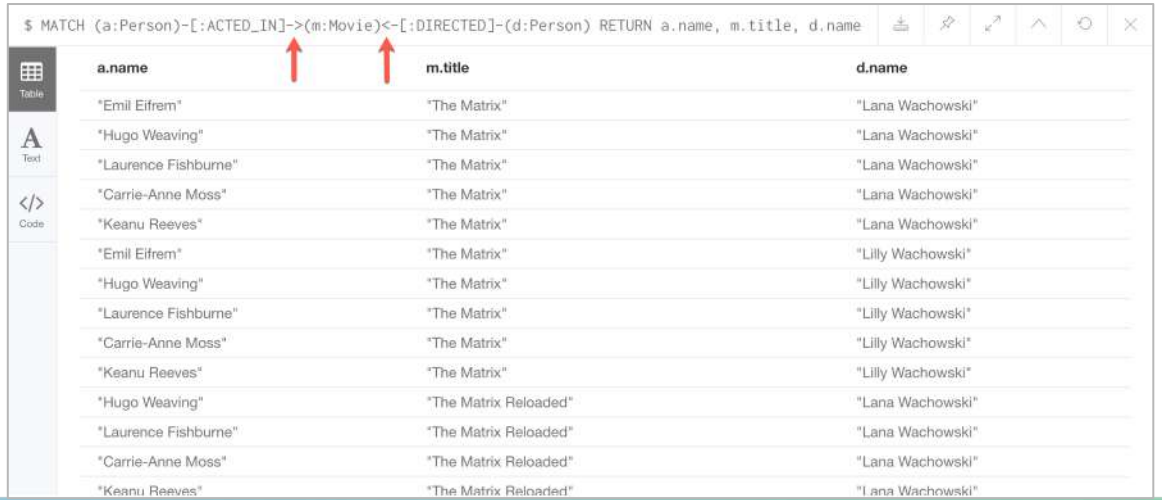
Sub-graph

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:

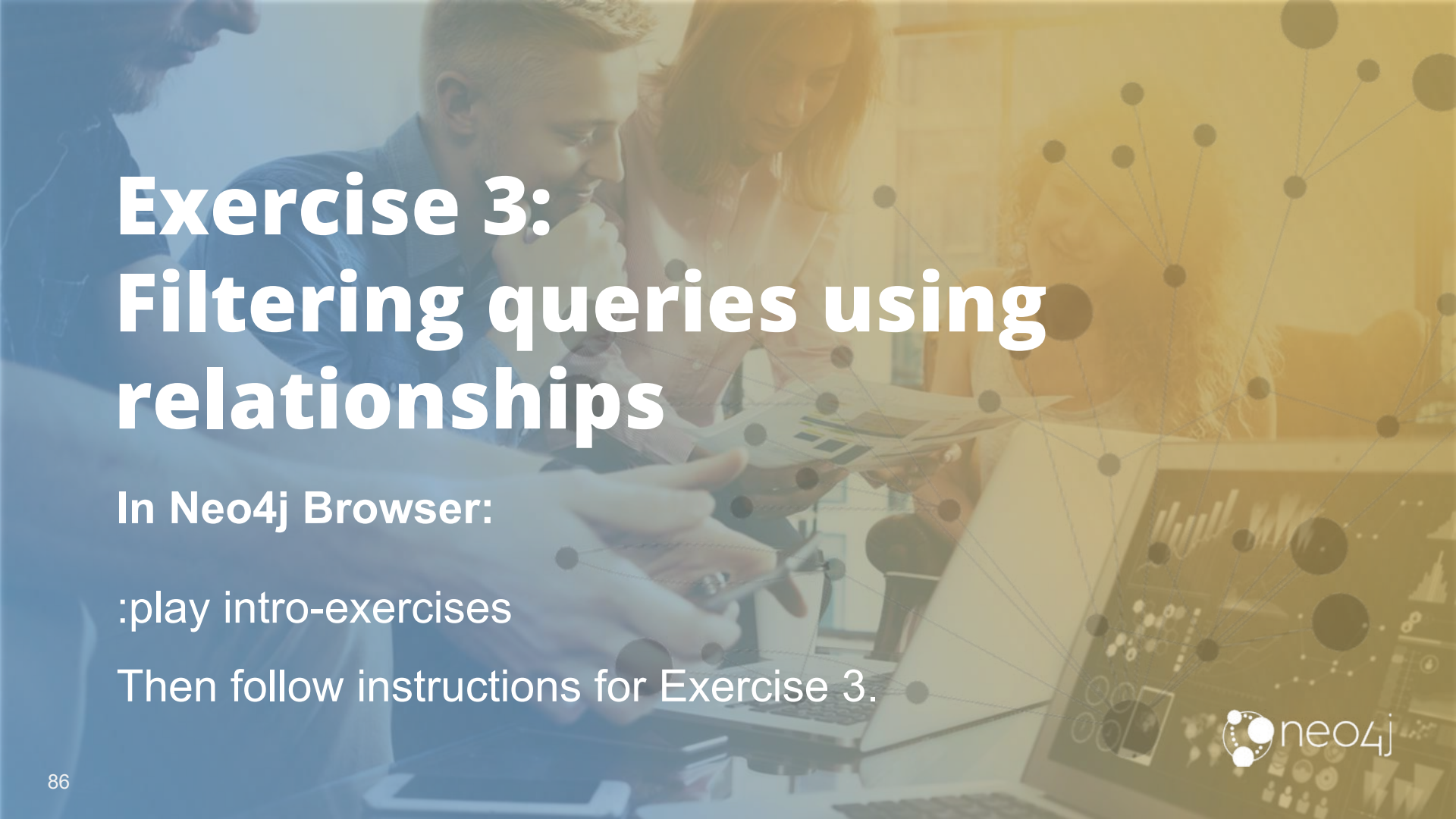
```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)
RETURN a.name, m.title, d.name
```



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).



Exercise 3: Filtering queries using relationships

In Neo4j Browser:

:play intro-exercises

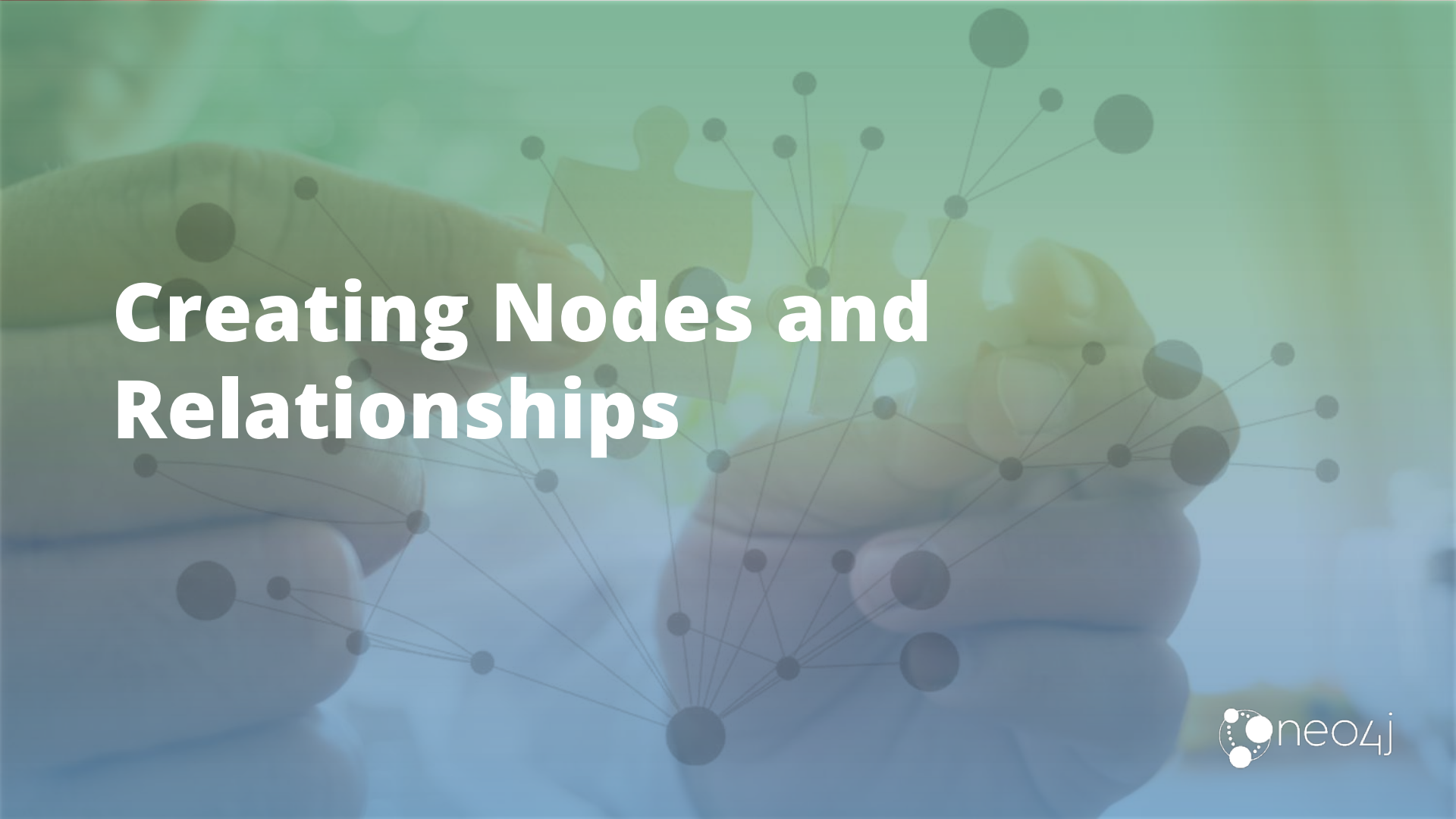
Then follow instructions for Exercise 3.

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.

Creating Nodes and Relationships



Creating a node

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

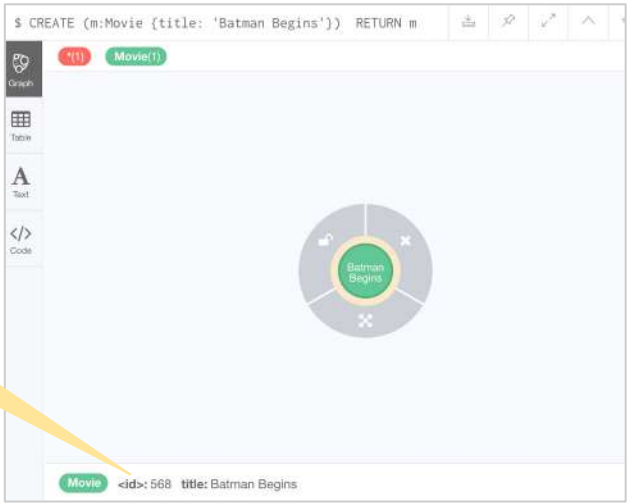
```
CREATE (:Movie {title: 'Batman Begins'})
```

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
```

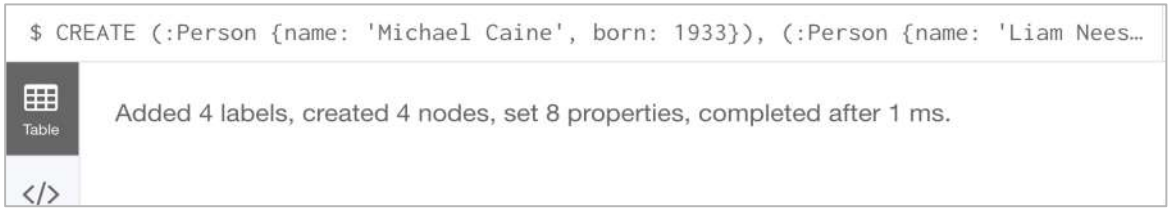


<id> is set by the graph engine

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
```

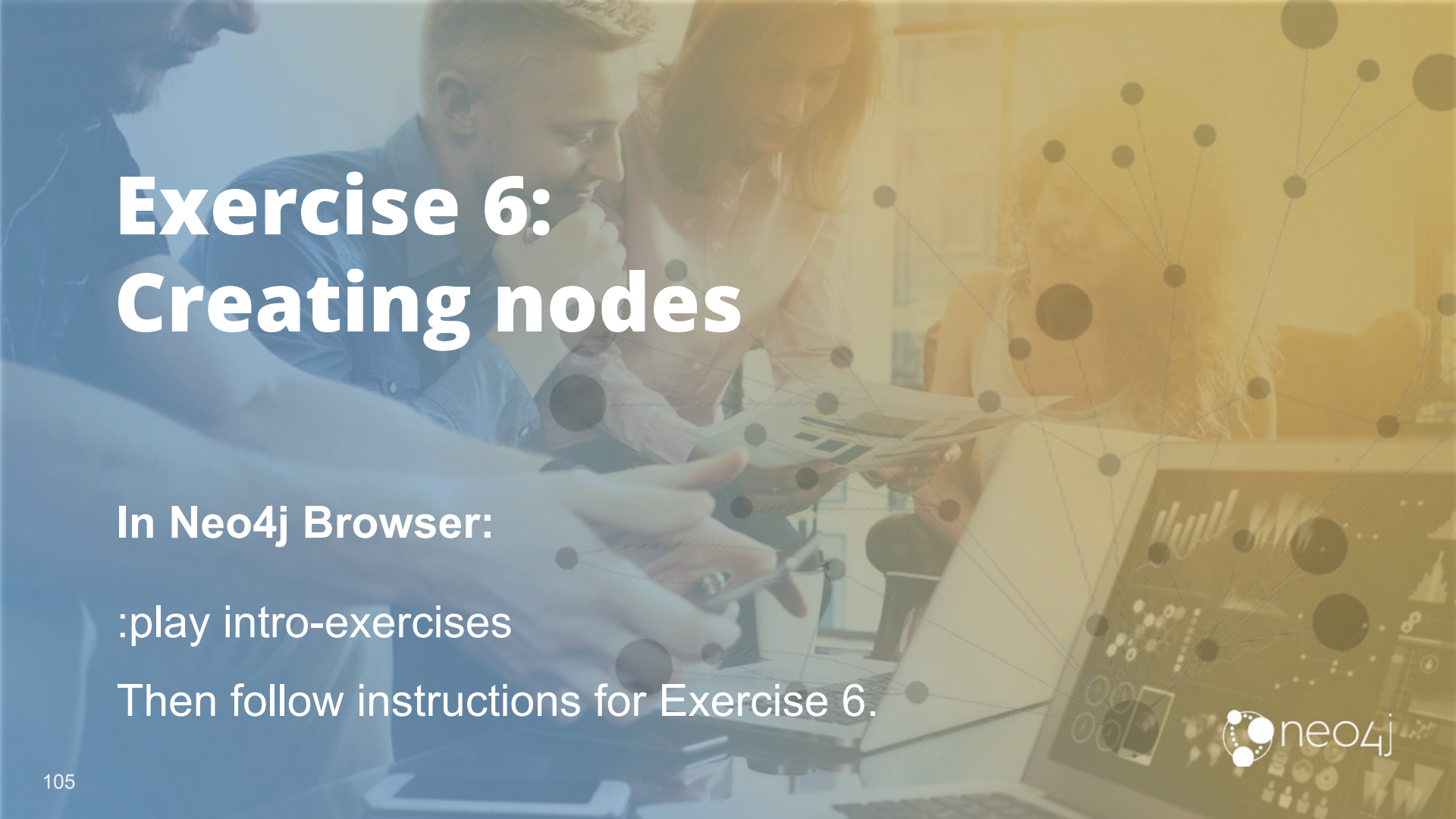
Removing properties from a node

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

In Neo4j Browser:

:play intro-exercises

Then follow instructions for Exercise 6.

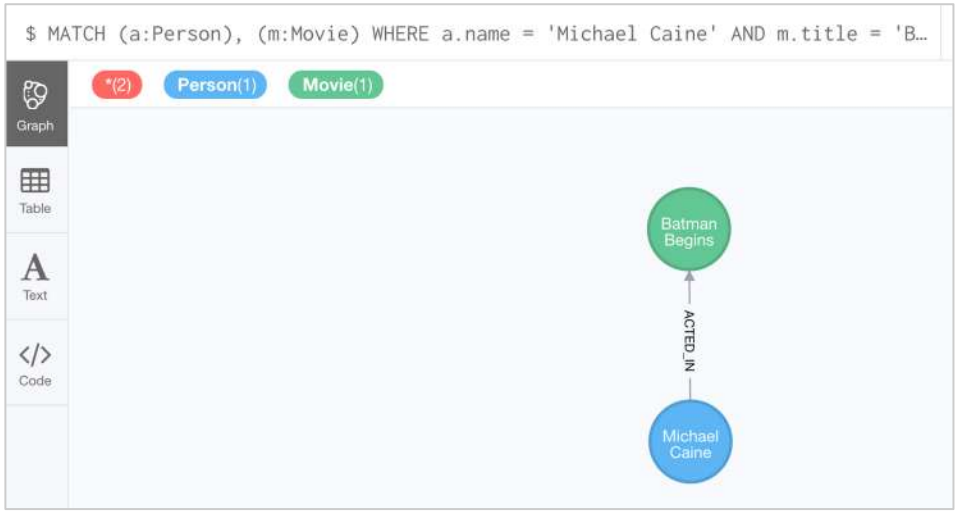
Creating a relationship

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*:

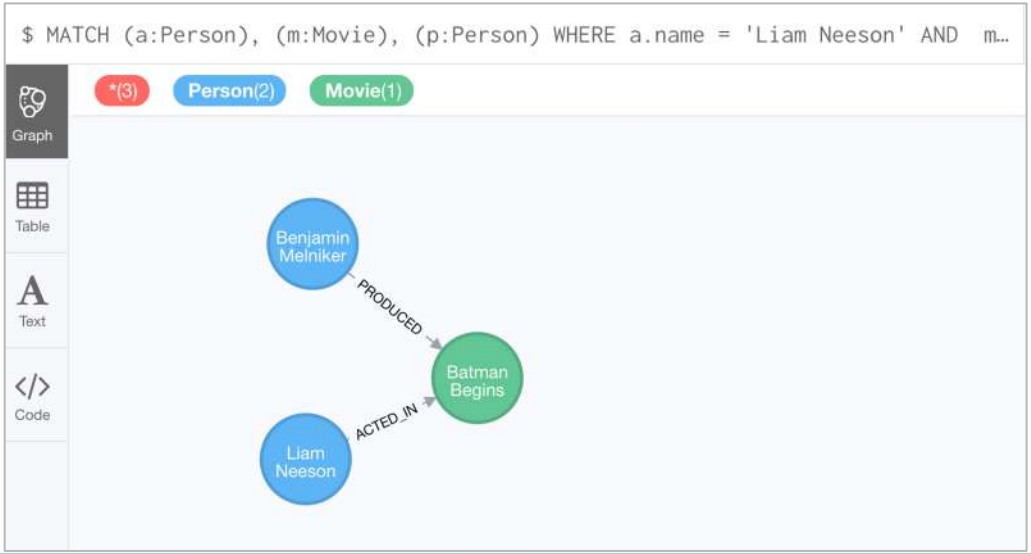
```
MATCH (a:Person), (m:Movie)
WHERE a.name = 'Michael Caine' AND
      m.title = 'Batman Begins'
CREATE (a)-[:ACTED_IN]->(m)
RETURN a, m
```



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
```



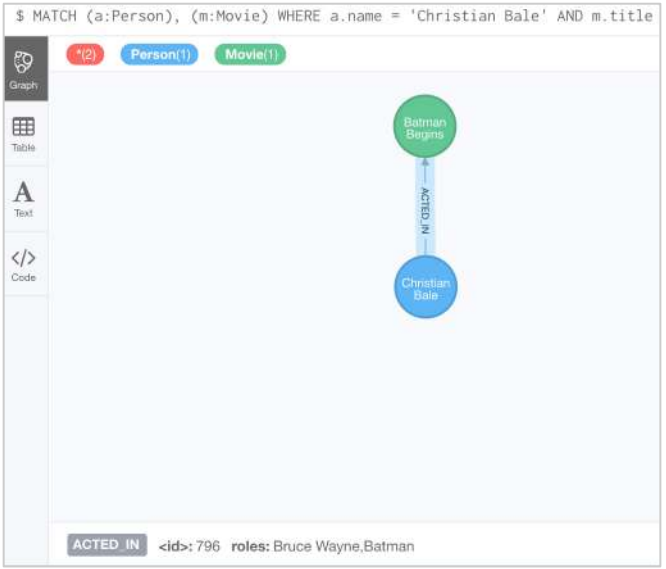
neo4j

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
```

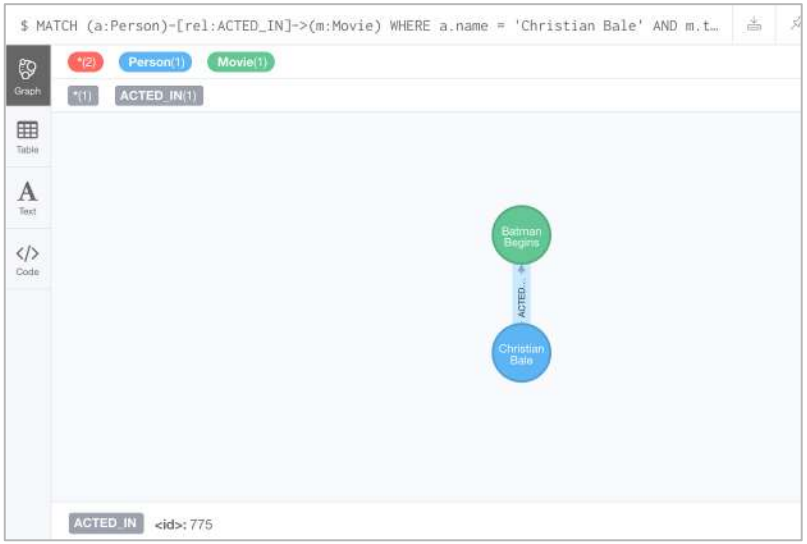


Removing properties from relationships

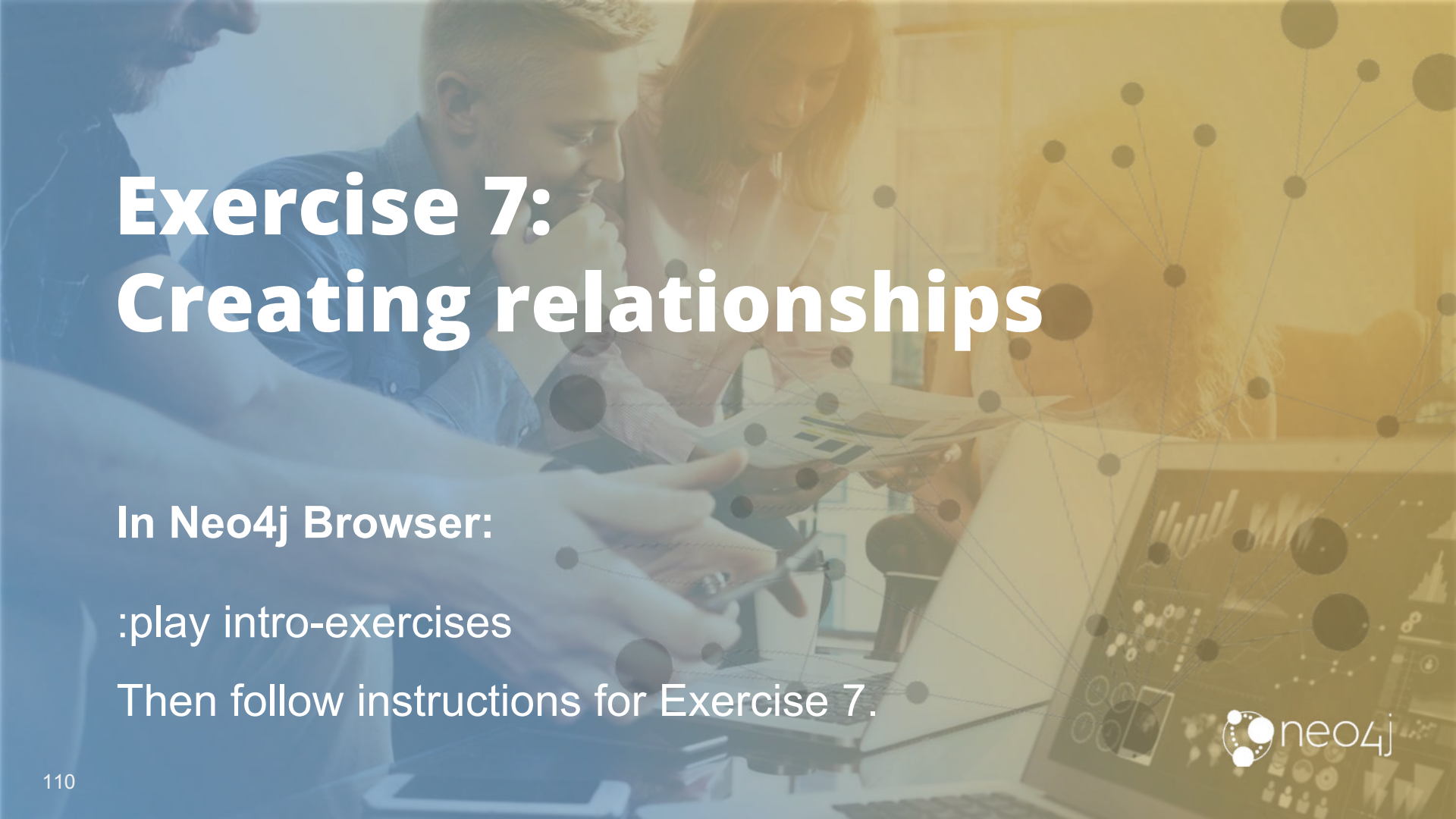
Same technique you use for removing node properties.

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
```



neo4j



Exercise 7: Creating relationships

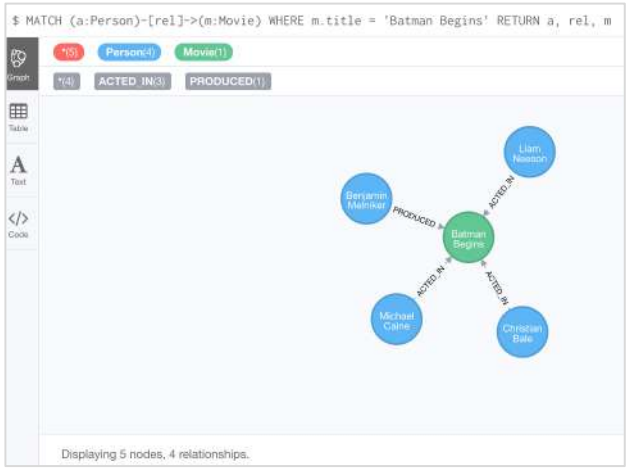
In Neo4j Browser:

:play intro-exercises

Then follow instructions for Exercise 7.

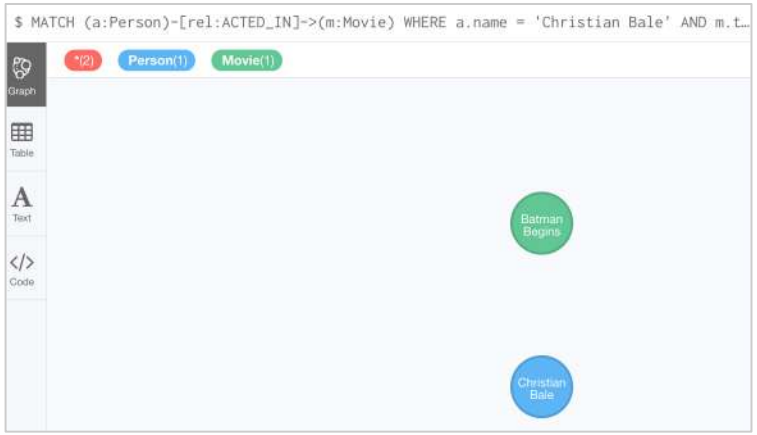
Deleting a relationship

Batman Begins relationships:



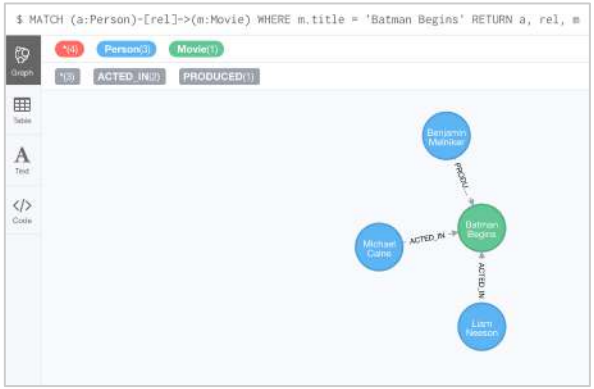
Delete the *:ACTED_IN* relationship between *Christian Bale* and *Batman Begins*:

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

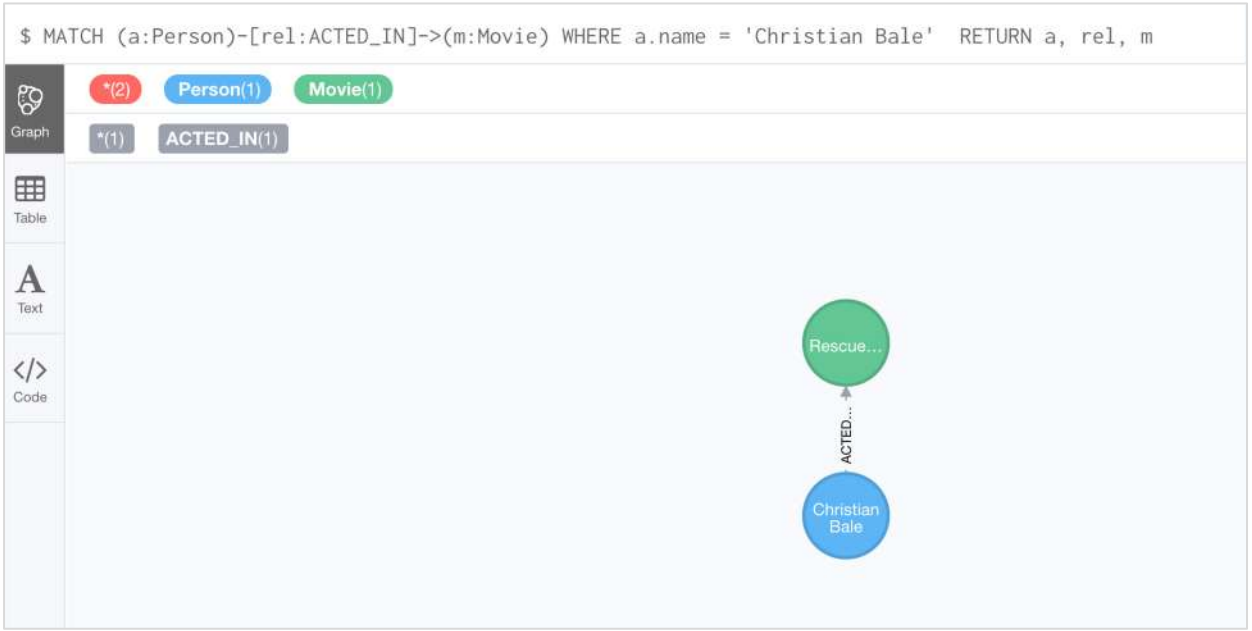


After deleting the relationship from *Christian Bale* to *Batman Begins*

Batman Begins relationships:



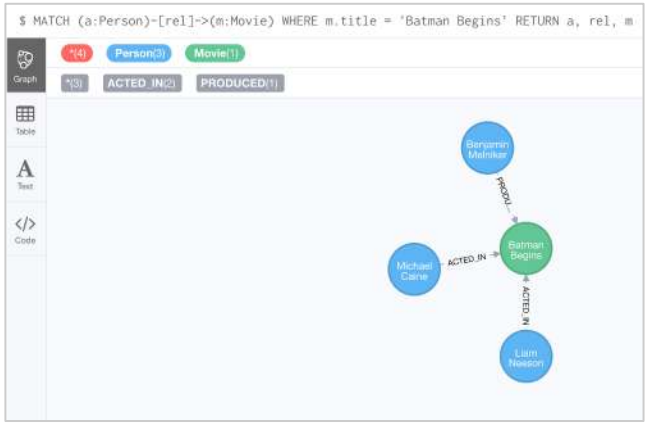
Christian Bale relationships:



neo4j

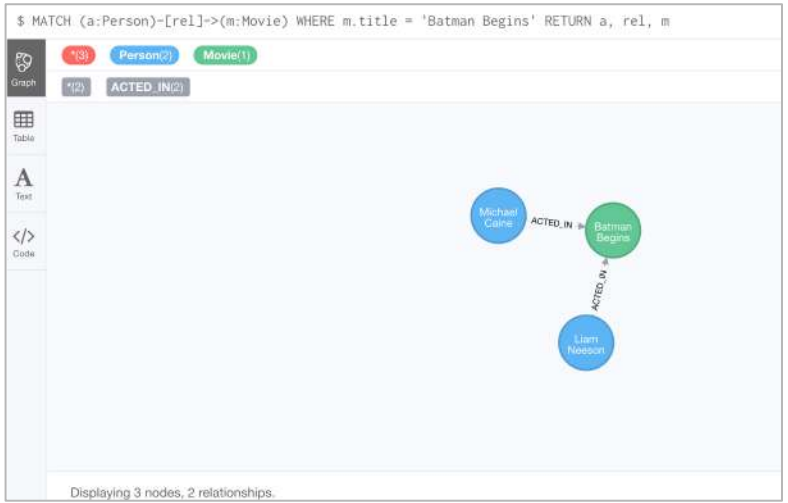
Deleting a relationship and a node - 1

Batman Begins relationships:



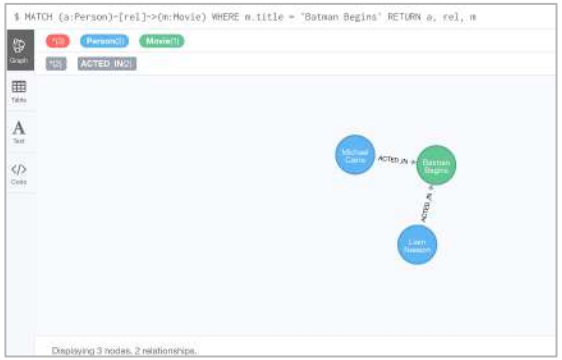
Delete the *:PRODUCED* relationship between *Benjamin Melniker* and *Batman Begins,* as well as the *Benjamin Melniker* node:

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



Deleting a relationship and a node - 2

Batman Begins relationships:

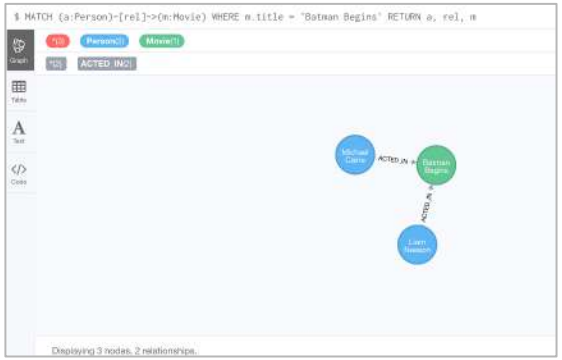


Attempt to delete *Liam Neeson* and <u>not</u> his relationships to any other nodes:

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

Deleting a relationship and a node - 3

Batman Begins relationships:



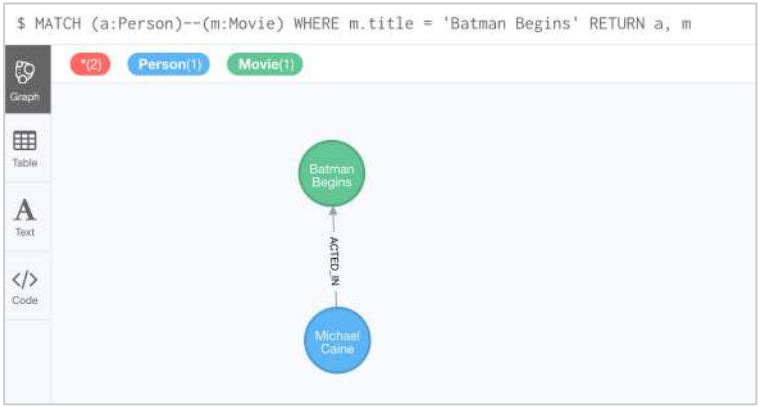
Delete *Liam Neeson* and his relationships to any other nodes:

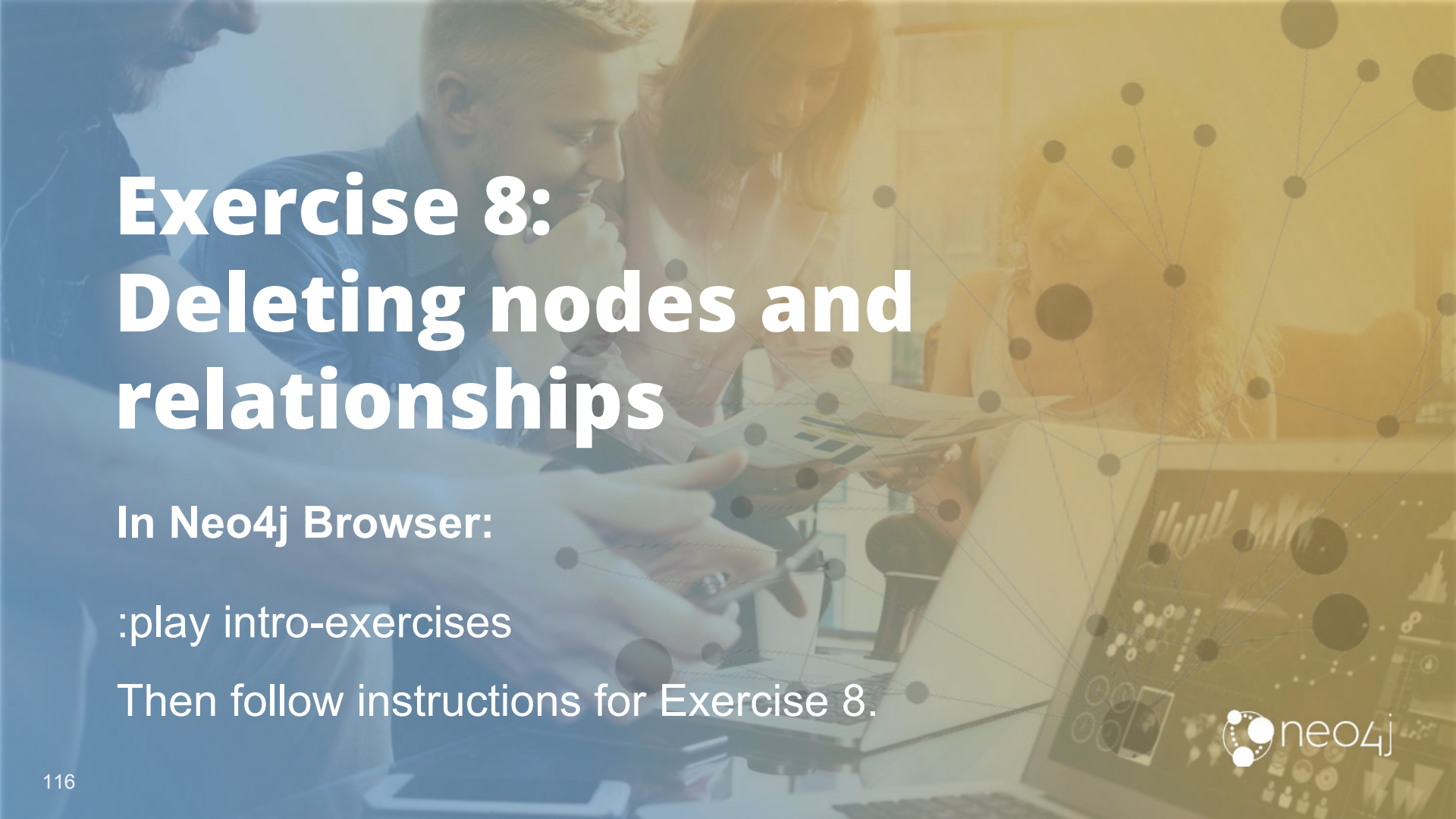
```
MATCH (p:Person)
WHERE p.name = 'Liam Neeson'
DETACH DELETE p
```

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

Table

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





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

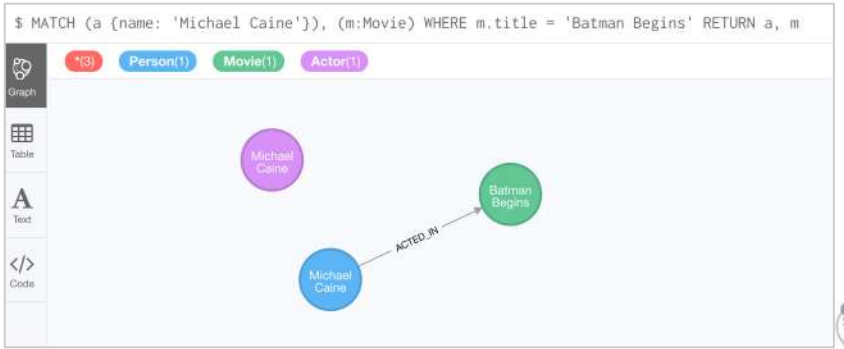
Current *Michael Caine Person* node:

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*:



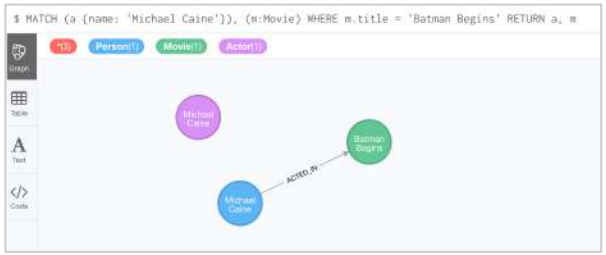
Using MERGE to create relationships

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

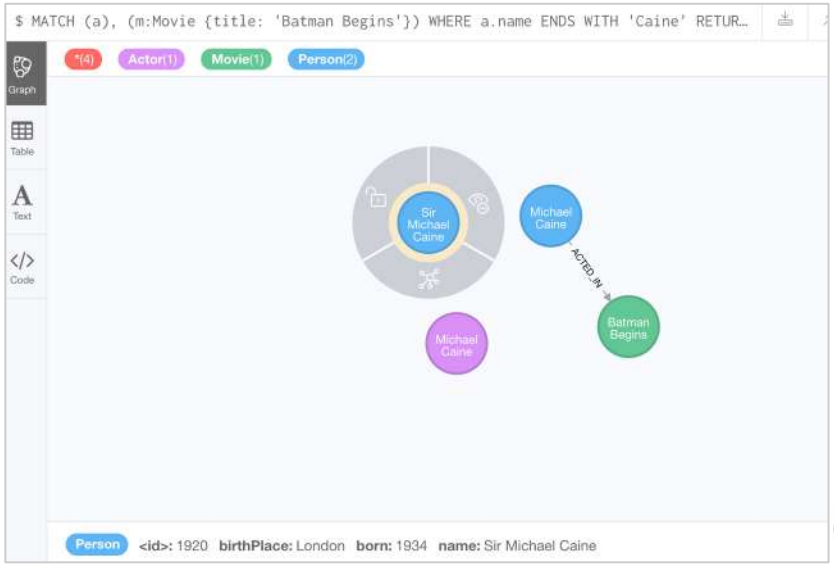
Current *Michael Caine* nodes:



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*:

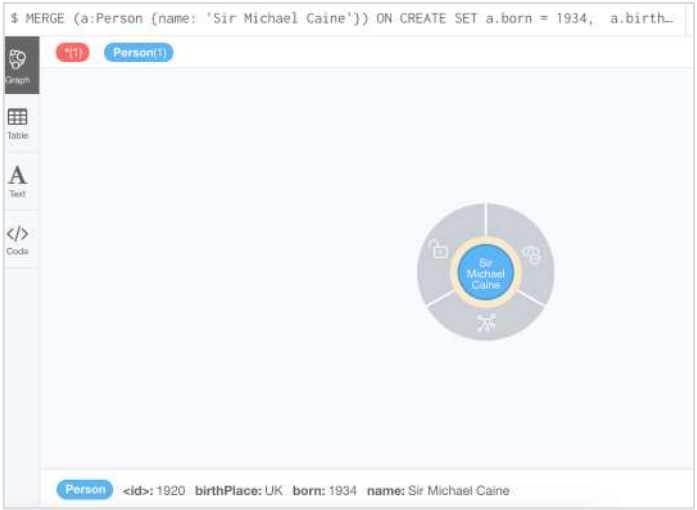


Specifying match behavior for the merge

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
```

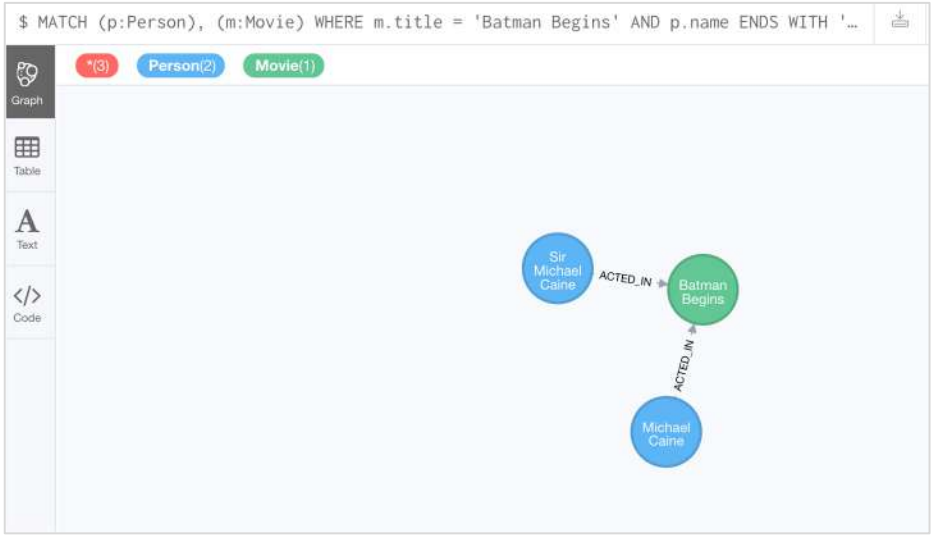


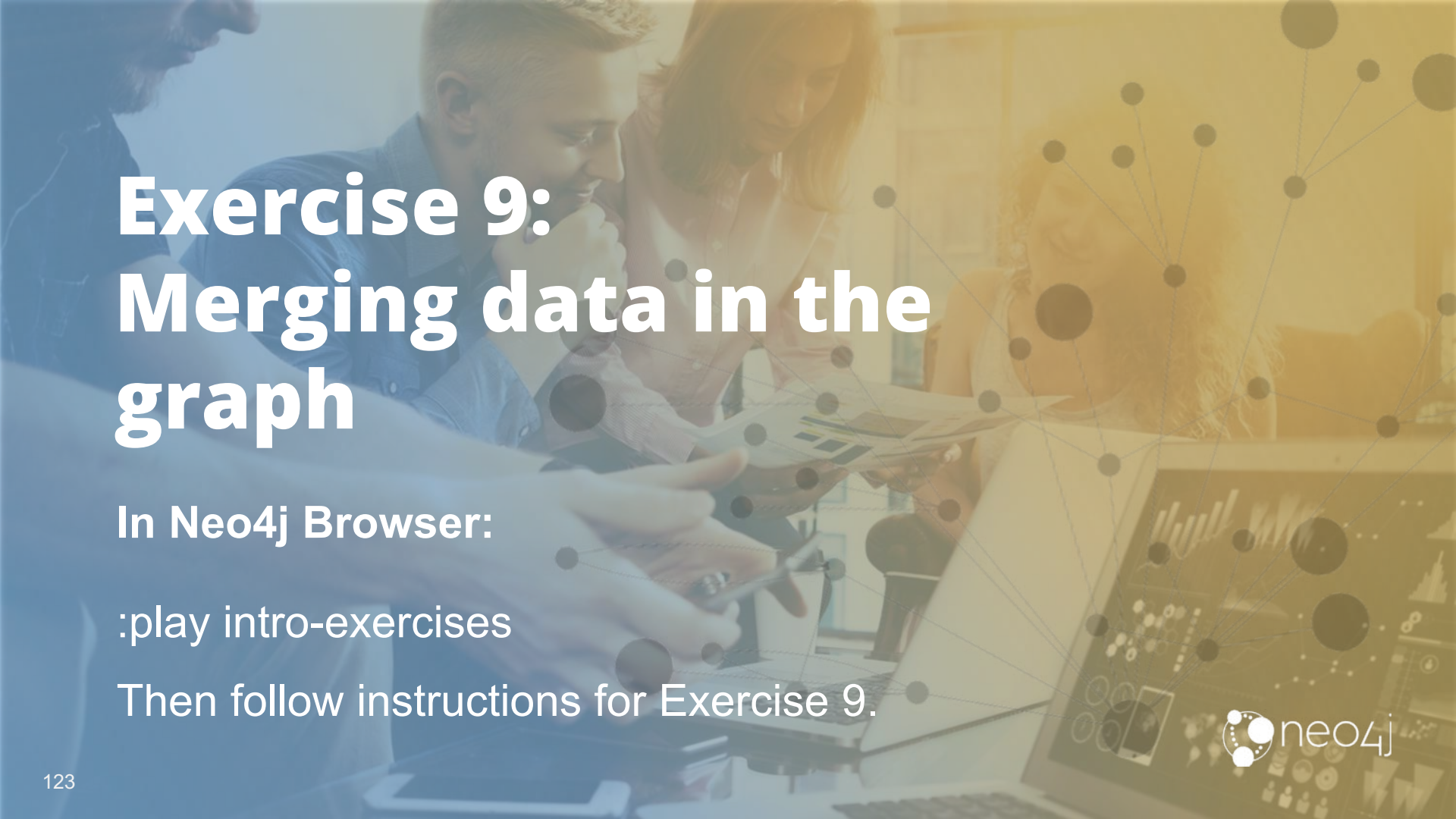
neo4j

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
```





Exercise 9: Merging data in the graph

In Neo4j Browser:

:play intro-exercises

Then follow instructions for Exercise 9.

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.

Indexes/Constraints

Indexes in Neo4j

- Neo4j supports two types of indexes on a node of a specific type:
  - single property
  - composite properties
- Indexes store redundant data that points to nodes with the specific property value or values.
- Unlike SQL, there is no such thing as a primary key in Neo4j. You can have multiple properties on nodes that must be unique.
- Add indexes before you create relationships between nodes.
- Creating an index on a property does not guarantee uniqueness.
  - But uniqueness and node key constraints are indexes that guarantee uniqueness.

When indexes are used

Single property indexes are used to determine the starting point for graph traversal using:

- Equality checks =
- Range comparisons >,>=,<, <=
- List membership IN
- String comparisons STARTS WITH, ENDS WITH, CONTAINS
- Existence checks exists()
- Spatial distance searches distance()
- Spatial bounding searches point()

Note: Composite indexes are only used for equality checks and list membership.

Creating a single-property index

Create a single-property index on the *released* property of all nodes of type *Movie*:

```
CREATE INDEX FOR (m:Movie) ON (m.released)
```

Creating a composite index - 1

Suppose first that we added the property, *videoFormat* to every *Movie* node and set its value, based upon the released date of the movie as follows:

```
MATCH (m:Movie)
WHERE m.released >= 2000
SET m.videoFormat = 'DVD';
MATCH (m:Movie)
WHERE m.released < 2000
SET m.videoFormat = 'VHS'
```

All *Movie* nodes in the graph now have both a *released* and *videoFormat* property.

Creating a composite index - 2

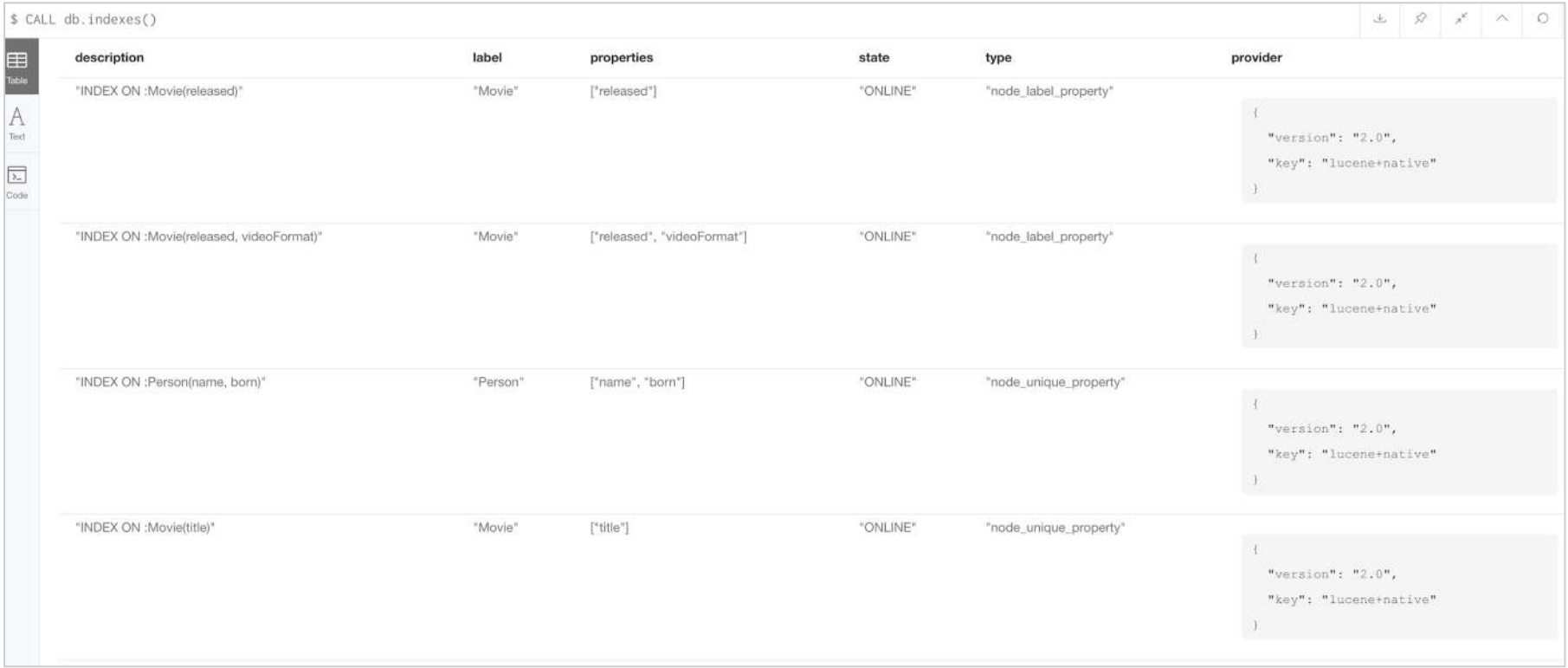
Create a composite index for every *Movie* node that uses the *videoFormat* and *released* properties:

```
CREATE INDEX FOR (m:Movie) ON (m.released, m.videoFormat)
```

Note: You can create a composite index with many properties.

Retrieving indexes

```
CALL db.indexes()
```



neo4j

Dropping indexes

```
DROP INDEX index_name
```

The name of the index can be found using the **SHOW INDEXES command**, given in the output column name

```
neo4j$ DROP INDEX index_5762eea0;
```

Removed 1 index, completed after 5 ms.

Loading Data

Importing data

- LOAD CSV - CSV datasets up to a few tens of millions of rows
- Neo4j-admin import - CSV files with data sizes in TBs - one time batch build
- Neo4j ETL tool - connection to RDMBS sources
- APOC - many options to sync data in and out of Neo4j e.g. JSON, Elastic Search, MongoDB, HIVE, Spark
- HOP - an open source ETL tool that has many connectors already pre-built (useful when gathering information from a large number of sources)
- Steaming - e.g. Kafka
- Spark
- Stored Procedures - Java based, can create custom integration to any service
- API/Driver - can create custom integration to any service in preferred language Java/Python/Go etc

Importing data

CSV import is commonly used to import data into a graph where you can:

- Load data from a URL (http(s) or file).
- Process data as a stream of records.
- Create or update the graph with the data being loaded.
- Use transactions during the load.
- Transform and convert values from the load stream.
- Load up to 10M nodes and relationships.

Steps for importing data

1. Determine the number of lines that will be loaded.
   - Is the load possible without special processing to handle transactions?
2. Examine the data and see if it may need to be reformatted.
   - Does data need alterations based upon your data requirements?
3. Make sure reformatting you will do is correct.
   - Examine final formatting of data before loading it.
4. Load the data and create nodes in the graph.
5. Load the data and create the relationships in the graph.

Importing normalized data - 1

Example CSV file, **movies_to_load.csv**:

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

1. Determine the number of lines that will be loaded:

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

Importing normalized data - 2

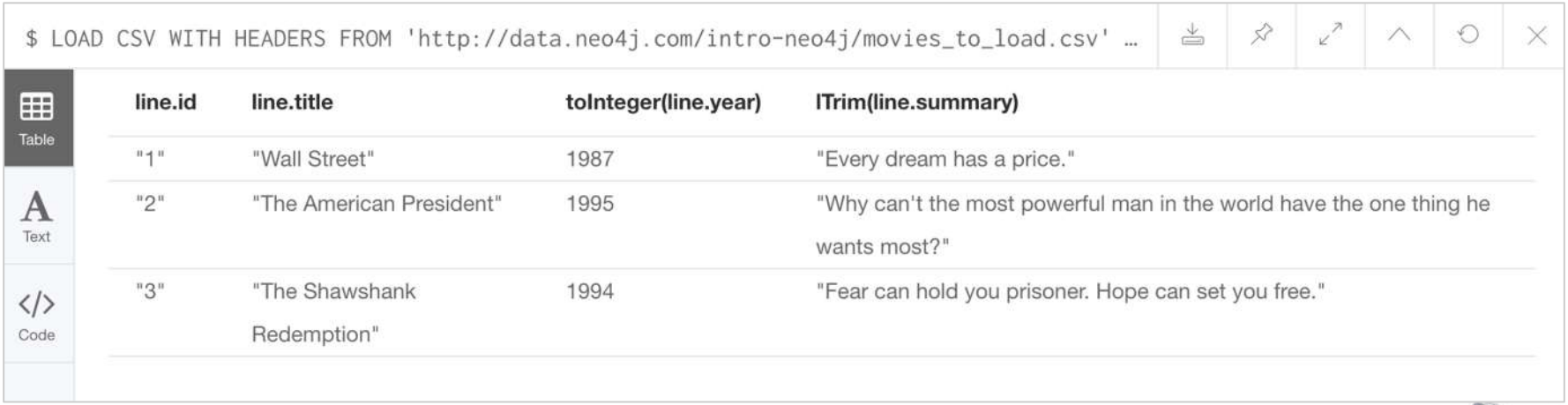
2. Examine the data and see if it may need to be reformatted:

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

Importing normalized data - 3

3. Format the data prior to loading:

```
LOAD CSV WITH HEADERS
FROM 'http://data.neo4j.com/intro-neo4j/movies_to_load.csv'
AS line
RETURN line.id, line.title, toInteger(line.year), trim(line.summary)
```



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

Importing normalized data - 4

4. Load the data and create the nodes in the graph:

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/intro-neo4j/movies_to_load.csv'
AS line
CREATE (movie:Movie { movieId: line.id,
                      title: line.title,
                      released: toInteger(line.year) ,
                      tagline: trim(line.summary)})
```

Importing the Person data

Example CSV file, **persons_to_load.csv**:

```
Id,name,birthyear
1,Charlie Sheen, 1965
2,Oliver Stone, 1946
3,Michael Douglas, 1944
4,Martin Sheen, 1940
5,Morgan Freeman, 1937
```

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/intro-neo4j/persons_to_load.csv'
AS line
MERGE (actor:Person { personId: line.Id })
ON CREATE SET actor.name = line.name,
              actor.born = toInteger(trim(line.birthyear))
```

We use MERGE to ensure that we will not create any duplicate nodes

Creating the relationships

Example CSV file, **roles_to_load.csv**:

```
personId,movieId,role
1,1,Bud Fox
4,1,Carl Fox
3,1,Gordon Gekko
4,2,A.J. MacInerney
3,2,President Andrew Shepherd
5,3,Ellis Boyd 'Red' Redding
```

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/intro-neo4j/roles_to_load.csv'
AS line
MATCH (movie:Movie { movieId: line.movieId })
MATCH (person:Person { personId: line.personId })
CREATE (person)-[:ACTED_IN { roles: [line.role]}]->(movie)
```

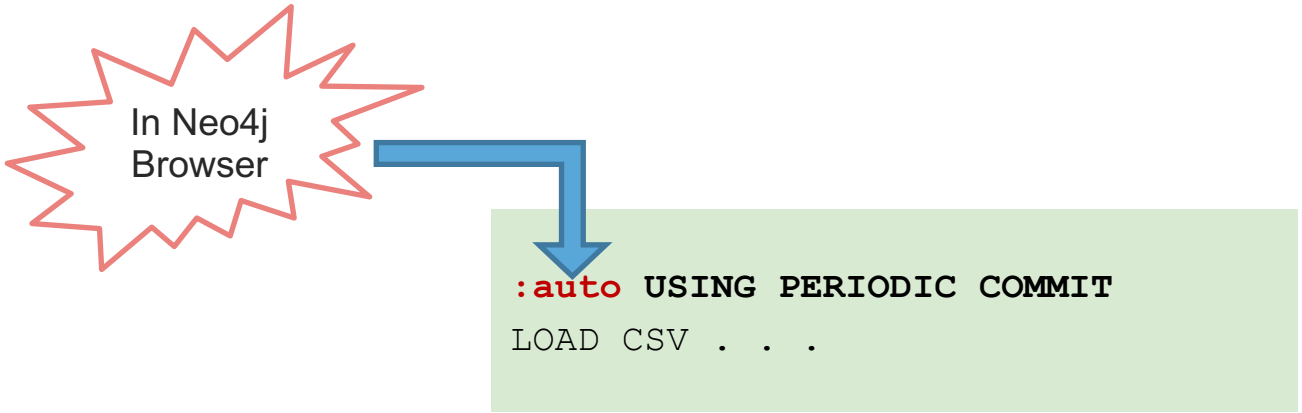
Importing denormalized data

Example CSV file, **movie_actor_roles_to_load.csv**:

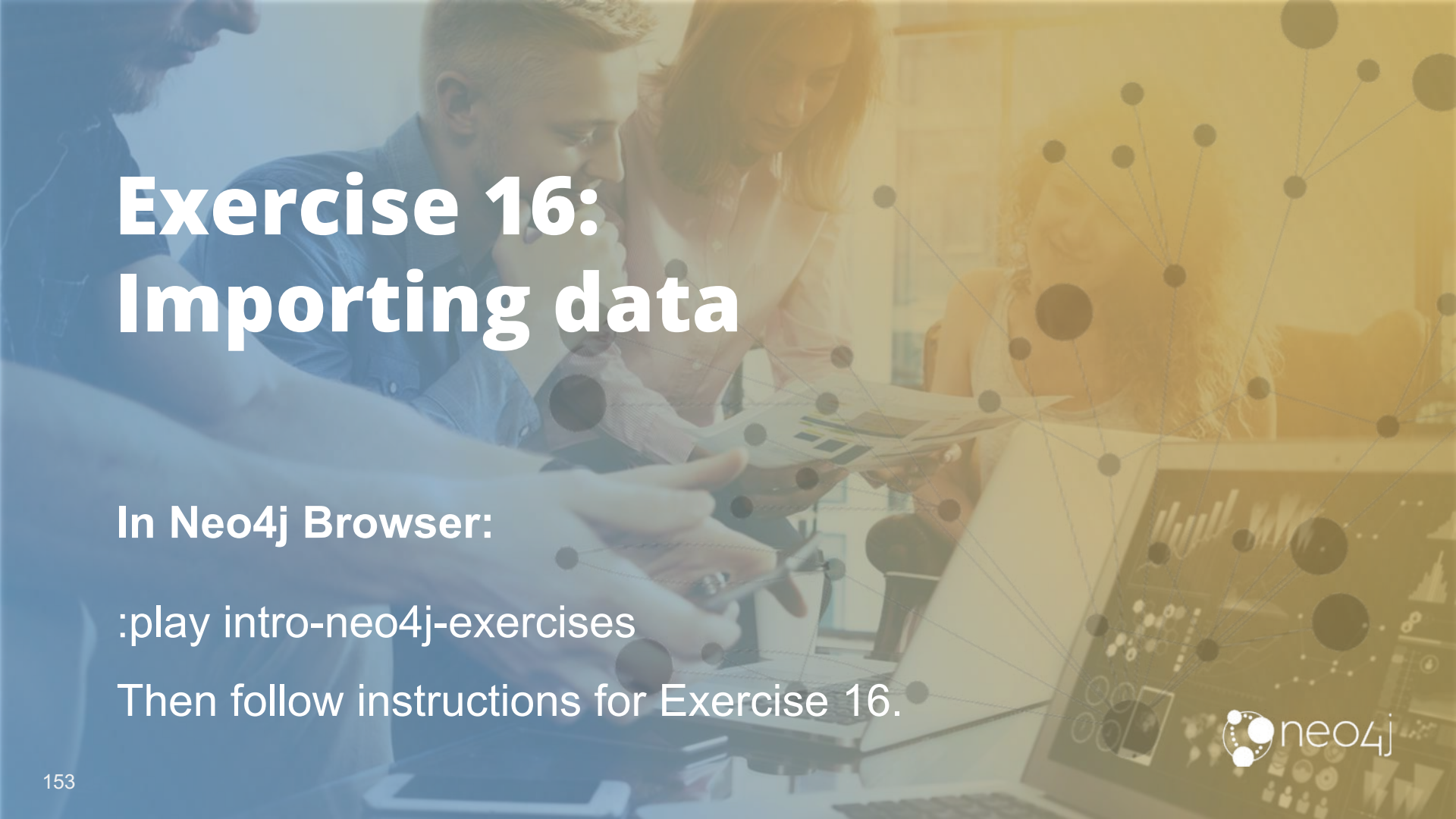
```
title;released;summary;actor;birthyear;characters
Back to the Future;1985;17 year old Marty McFly got home early last night. 30 years early.;Michael J. Fox;1961;Marty McFly
Back to the Future;1985;17 year old Marty McFly got home early last night. 30 years early.;Christopher Lloyd;1938;Dr. Emmet Brown
```

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/intro-neo4j/movie_actor_roles_to_load.csv'
AS line FIELDTERMINATOR ';'
MERGE (movie:Movie { title: line.title })
ON CREATE SET movie.released = toInteger(line.released),
              movie.tagline = line.summary
MERGE (actor:Person { name: line.actor })
ON CREATE SET actor.born = toInteger(line.birthyear)
MERGE (actor)-[r:ACTED_IN]->(movie)
ON CREATE SET r.roles = split(line.characters,',')
```

Importing a large dataset



Benefit: The graph engine will automatically commit data to avoid memory issues.



Exercise 16: Importing data

In Neo4j Browser:

:play intro-neo4j-exercises

Then follow instructions for Exercise 16.

Getting More Out of Queries

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:

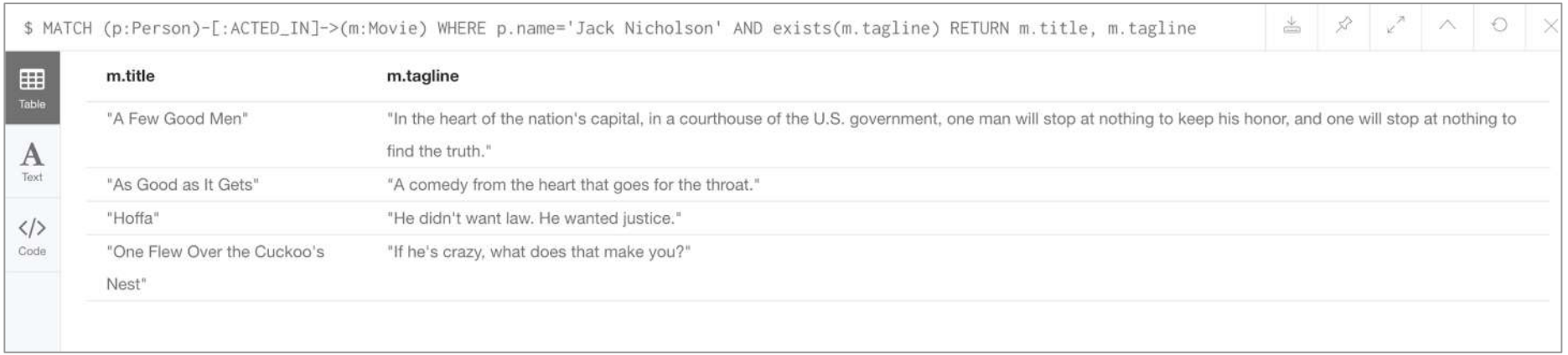
```
MATCH (p)
WHERE p:Person
RETURN p.name
```

```
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
```

Testing with patterns - 2

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

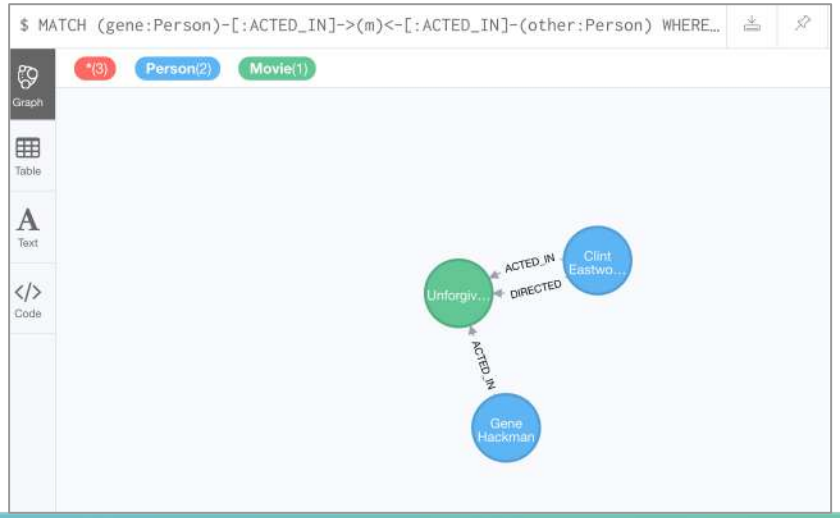
```
MATCH (p:Person)-[:WROTE]->(m:Movie)
WHERE r.year=1990
//WHERE NOT exists( (p)-[:DIRECTED]->(m) )
RETURN p.name, m.title
```

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

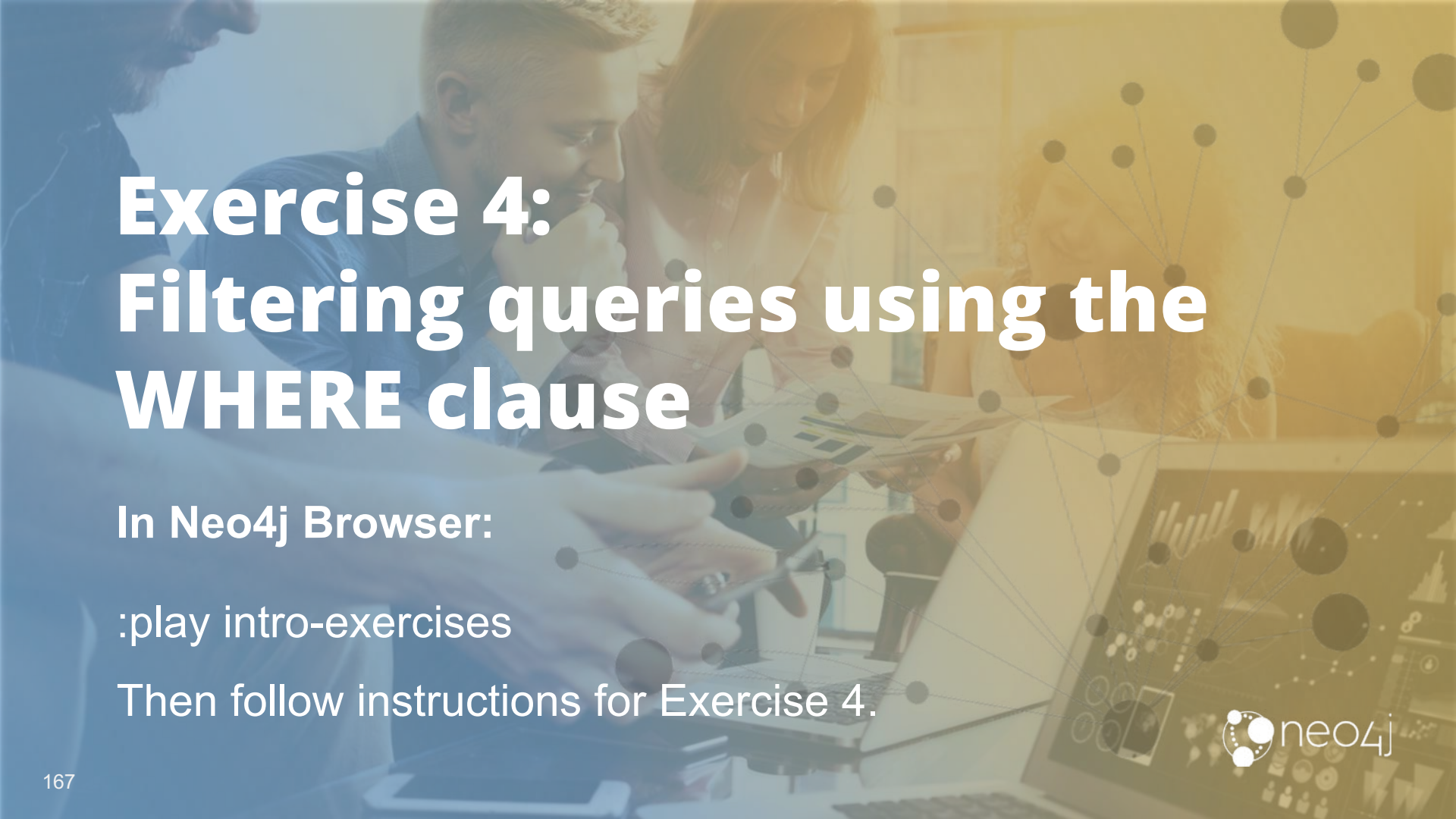
Find all people born in 1965 and 1970:

```
MATCH (p:Person)
WHERE p.born IN [1965, 1970, 1971, 1972]
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
```



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
```

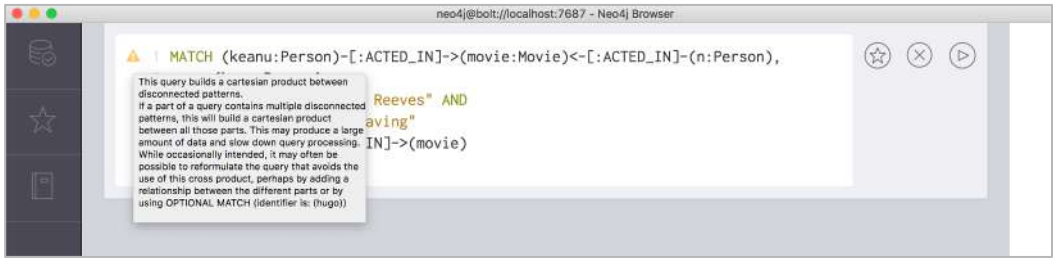
A best practice is to use a single MATCH pattern if possible:

```
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:

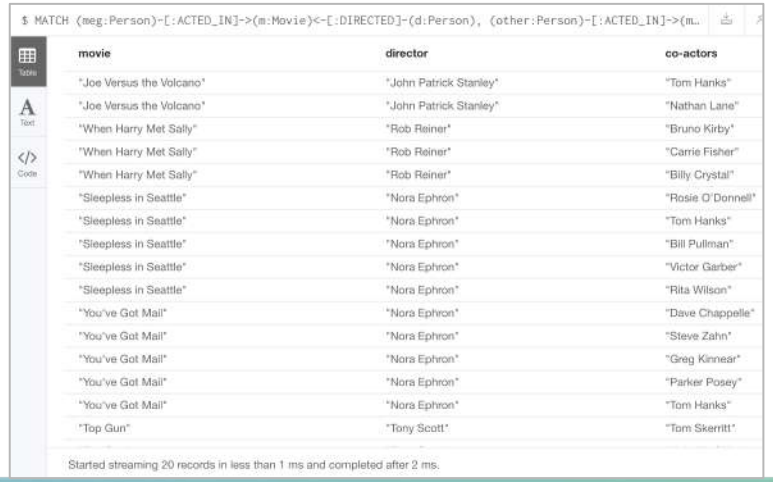
```
MATCH (keanu:Person)-[:ACTED_IN]->(movie:Movie)<-[:ACTED_IN]-(n:Person), (hugo:Person)
WHERE keanu.name='Keanu Reeves' AND hugo.name='Hugo Weaving' AND
      NOT (hugo)-[:ACTED_IN]->(movie)
RETURN n.name
```



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`
```



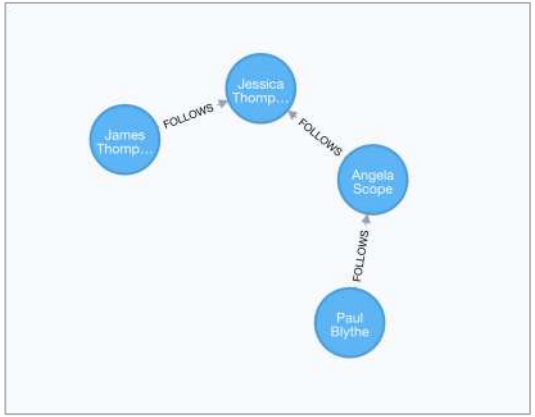
$ MATCH (meg:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person), (other:Person)-[:ACTED_IN]->(m...

Table | Text | Code

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



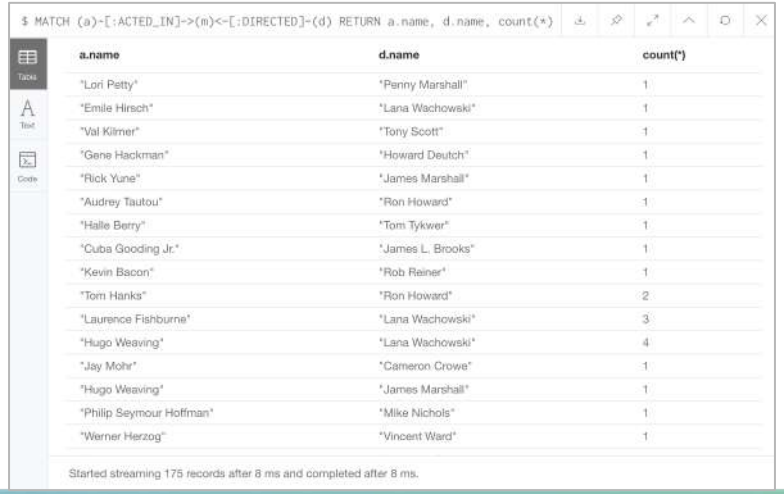
Find all people who are exactly two hops away from *Paul Blythe*:

```
MATCH (follower:Person)-[:FOLLOWS*..]->(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

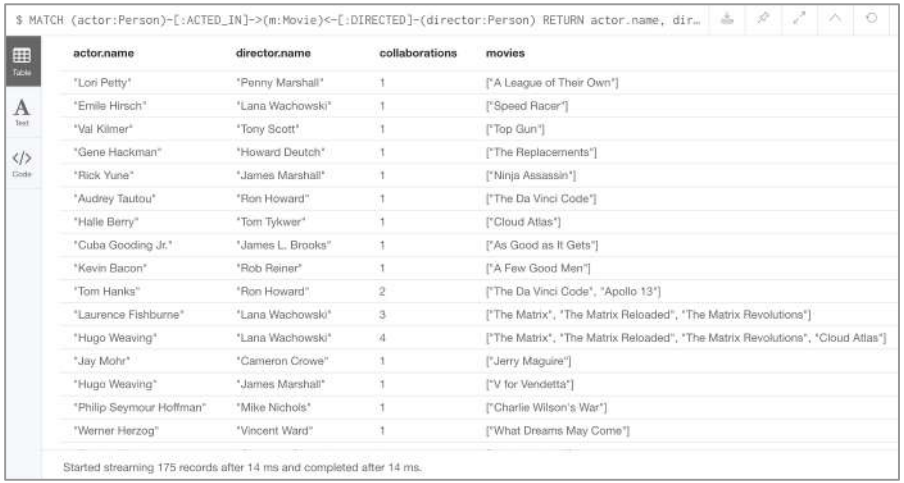
Find the movies that Tom Cruise acted in and return them as a list:

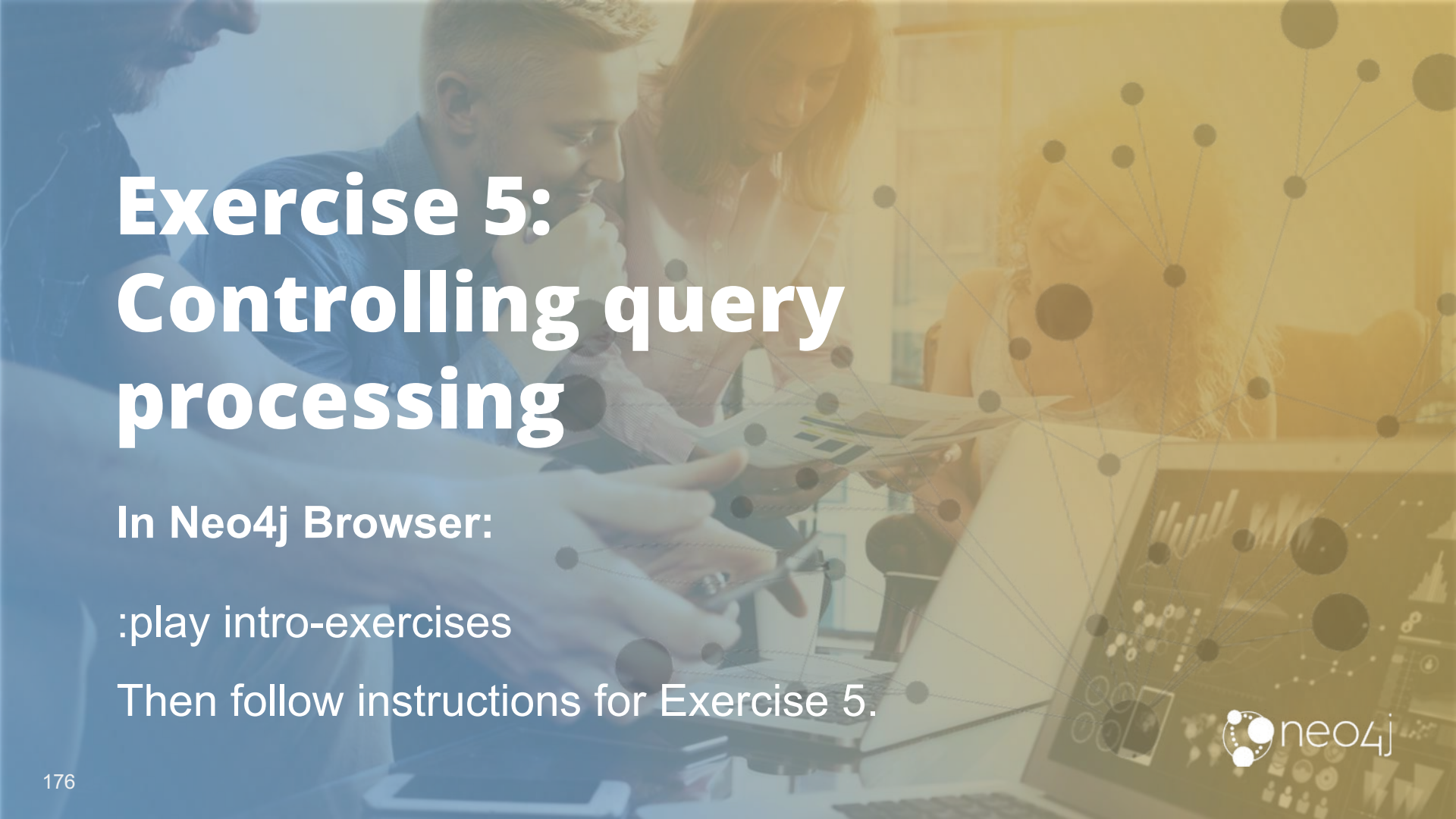
```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name ='Tom Cruise'
RETURN collect(m.title) AS `movies for Tom Cruise`
```

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

In Neo4j Browser:

:play intro-exercises

Then follow instructions for Exercise 5.

Getting More Out of Neo4j

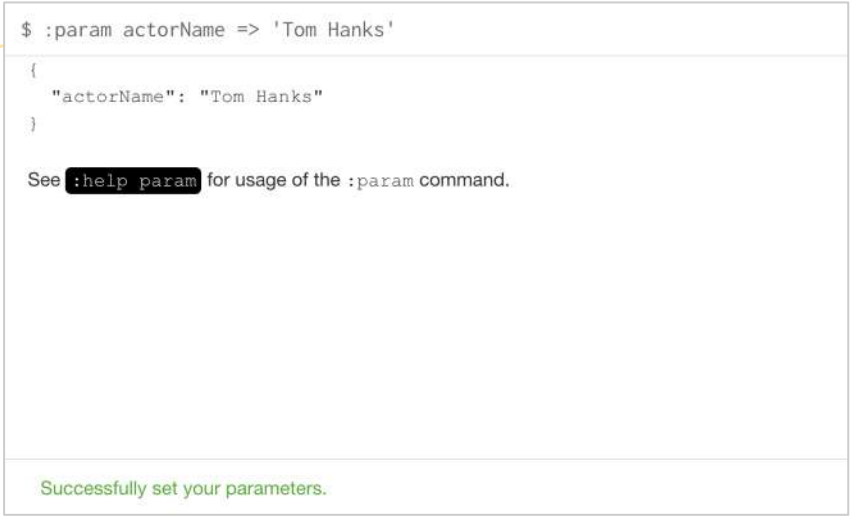
v 1.0

Cypher Parameters

Cypher parameters

Using Cypher parameters - 1

1. Set values for parameters in your Neo4j Browser session before you run the query.



2. Specify parameters using '$' in your Cypher query.

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

Using Cypher parameters - 2

When this query runs, *$actorName* has a value *Tom Hanks*:

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

Table | Text | Code

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

Using Cypher parameters - 3

Change the value of the parameter, *$actorName* to *Tom Cruise*:

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

Re-run the same query:

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

Table | Text | </>

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

Understanding Query Execution

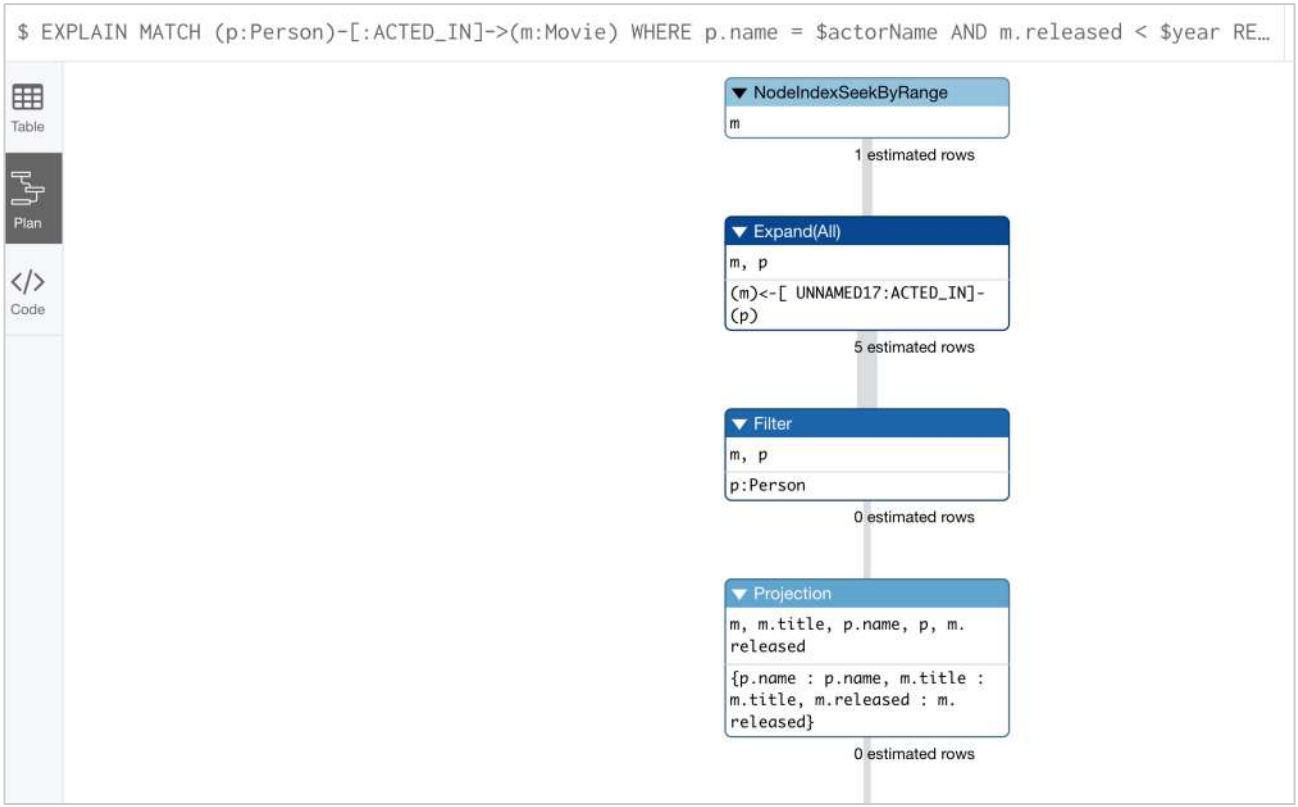
Analyzing Cypher queries - EXPLAIN - 1

- Provides information about the query plan.
- Does not execute the Cypher statement.

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

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

Analyzing Cypher queries - EXPLAIN - 2



Analyzing Cypher queries - PROFILE - 1

- Provides information about the query plan.
- Executes the Cypher statement.
- Provides information about db hits.

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

Analyzing Cypher queries - PROFILE - 2

Profile query where node labels are specified:

Profile query where node labels <u>not</u> are specified:

Monitoring queries

There are two reasons why a Cypher query may take a long time:

1. The query returns a lot of data. The query completes execution in the graph engine, but it takes a long time to create the result stream to return to the client (eg: Neo4j Browser).

```
MATCH (a)--(b)--(c)--(d)--(e)--(f) RETURN a
```

You should avoid these type of queries! You cannot monitor them.

1. The query takes a long time to execute in the graph engine.

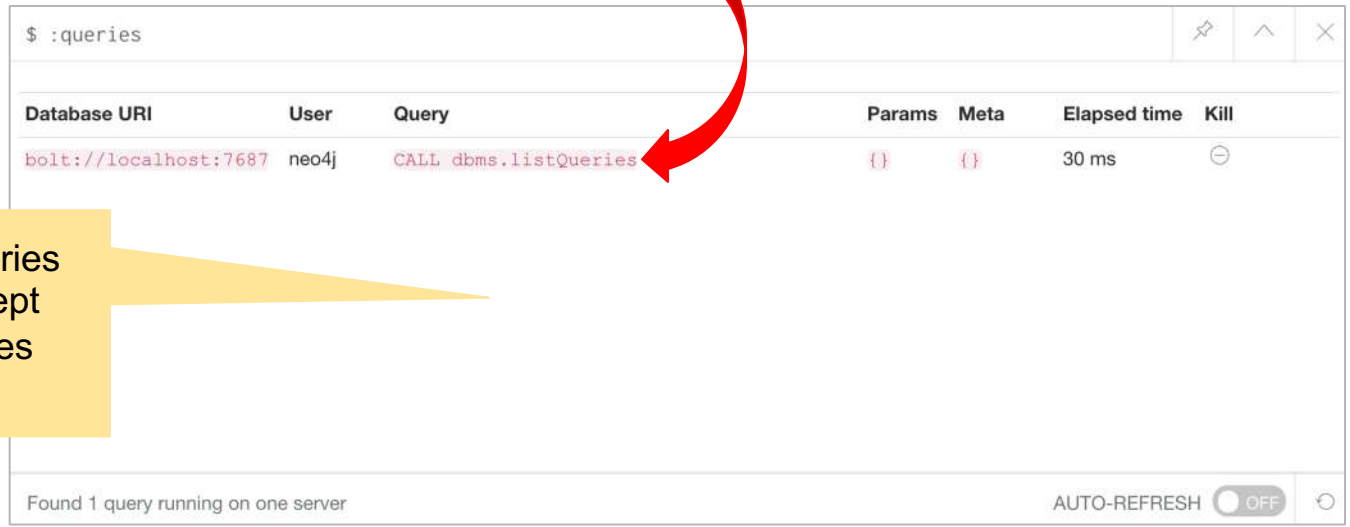
```
MATCH (a), (b), (c), (d), (e) RETURN count(id(a))
```

You can monitor and kill these types of queries.

Viewing running queries

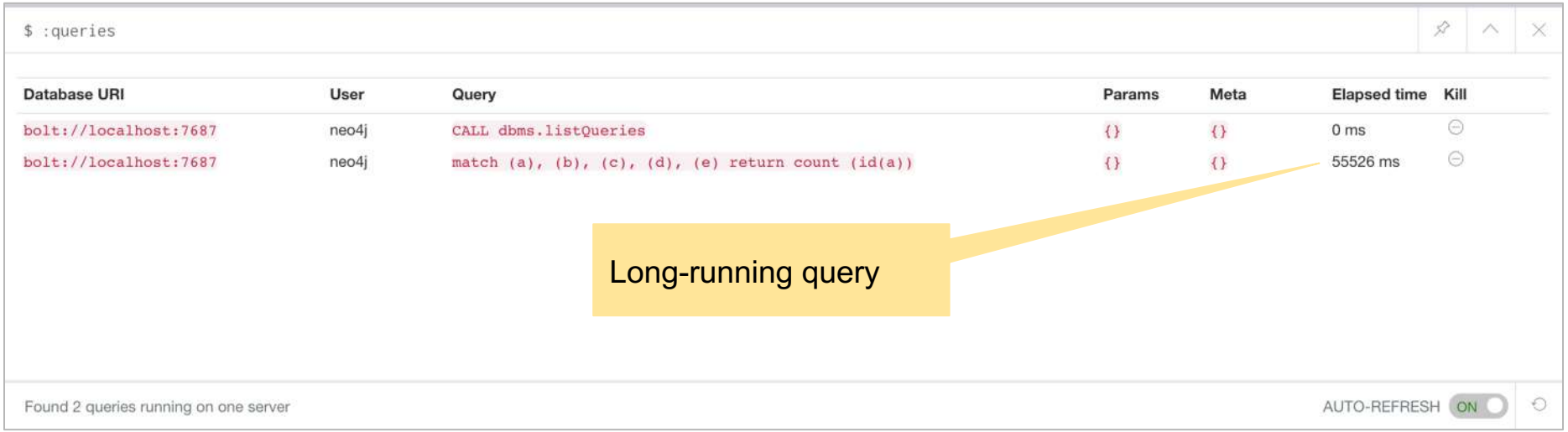
If your query is taking a long time to execute your first have to determine if it is running in the graph engine:

1. Open a new Neo4j Browser session.
2. Execute the **:queries** command.



No other queries running, except for the :queries command.

Viewing long-running queries



Killing long-running queries

Monitoring session

Query session

Handling “rogue” queries

If your query is taking a long time to execute and you cannot monitor it, your options are to:

1. Close the Neo4j Browser session that is stuck and start a new Neo4j Browser session.
2. If that doesn’t work:
   a. On Neo4j Desktop, restart the database.
   b. In Neo4j Sandbox, shut down the sandbox (ouch!). You need to re-create the Sandbox.

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/.