Graph DBs and Neo4J (Part II)

NOTES:
1. Many thanks to Neo4J for providing this material and granting us permission to use it!
2. These slides have been lightly modified for CS122D use.
Cypher: Neo4J’s Query Language (Continued)
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)

<table>
<thead>
<tr>
<th>p.name</th>
<th>type(rel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Emil Eifrem&quot;</td>
<td>&quot;ACTED_IN&quot;</td>
</tr>
<tr>
<td>&quot;Joel Silver&quot;</td>
<td>&quot;PRODUCED&quot;</td>
</tr>
<tr>
<td>&quot;Lana Wachowski&quot;</td>
<td>&quot;DIRECTED&quot;</td>
</tr>
<tr>
<td>&quot;Lilly Wachowski&quot;</td>
<td>&quot;DIRECTED&quot;</td>
</tr>
<tr>
<td>&quot;Hugo Weaving&quot;</td>
<td>&quot;ACTED_IN&quot;</td>
</tr>
<tr>
<td>&quot;Laurence Fishburne&quot;</td>
<td>&quot;ACTED_IN&quot;</td>
</tr>
<tr>
<td>&quot;Carrie-Anne Moss&quot;</td>
<td>&quot;ACTED_IN&quot;</td>
</tr>
<tr>
<td>&quot;Keanu Reeves&quot;</td>
<td>&quot;ACTED_IN&quot;</td>
</tr>
</tbody>
</table>

Started streaming 8 records after 1 ms and completed after 2 ms.
Retrieving properties for a relationship - 1

- Jessica Thompson
  - :Person
  - :REVIEWED -> :Movie
  - summary: ‘A solid romp’
  - rating: 68

- James Thompson
  - :Person
  - :REVIEWED -> :Movie
  - summary: ‘Fun, but a little far fetched’
  - rating: 65

- Movie
  - title: ‘The Da Vinci Code’
  - released: 2006
  - tagline: ‘Break the Codes’
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
```

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

<table>
<thead>
<tr>
<th>p.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;James Thompson&quot;</td>
</tr>
</tbody>
</table>
```
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
```

![Diagram showing nodes and relationships](image-url)
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

$ 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
```

![Graph representation of the query](image)
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
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
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).
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 <--

```cypher
MATCH (:Person {name: 'Diane Keaton'})-[movRel:ACTED_IN]->(:Movie {title:"Something's Gotta Give"})
RETURN movRel.roles
```
More Advanced Queries
Filtering queries using **WHERE**

Previously you retrieved nodes as follows:

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

A more flexible syntax for the same query is:

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

Testing more than equality:

```cypher
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
Indexing in Neo4J (for WHERE efficiency)

Consider an application with flights and airports:

We want any queries for a flight to be fast when the flight number is specified.

CREATE INDEX ON :Flight(number)
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
```

<table>
<thead>
<tr>
<th>m.title</th>
<th>m.tagline</th>
</tr>
</thead>
</table>
| "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 p.name STARTS WITH 'Michael' RETURN p.name

<table>
<thead>
<tr>
<th>p.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Michael Clarke Duncan&quot;</td>
</tr>
<tr>
<td>&quot;Michael Sheen&quot;</td>
</tr>
</tbody>
</table>

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

24
Testing with regular expressions

Find people whose name starts with **Tom**:

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

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

<table>
<thead>
<tr>
<th>p.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom Cruise&quot;</td>
</tr>
<tr>
<td>&quot;Tom Skerritt&quot;</td>
</tr>
<tr>
<td>&quot;Tom Hanks&quot;</td>
</tr>
<tr>
<td>&quot;Tom Tykwer&quot;</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>p.name</th>
<th>m.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Aaron Sorkin&quot;</td>
<td>&quot;A Few Good Men&quot;</td>
</tr>
<tr>
<td>&quot;Jim Cash&quot;</td>
<td>&quot;Top Gun&quot;</td>
</tr>
<tr>
<td>&quot;Cameron Crowe&quot;</td>
<td>&quot;Jerry Maguire&quot;</td>
</tr>
<tr>
<td>&quot;Nora Ephron&quot;</td>
<td>&quot;When Harry Met Sally&quot;</td>
</tr>
<tr>
<td>&quot;David Mitchell&quot;</td>
<td>&quot;Cloud Atlas&quot;</td>
</tr>
<tr>
<td>&quot;Lilly Wachowski&quot;</td>
<td>&quot;V for Vendetta&quot;</td>
</tr>
<tr>
<td>&quot;Lana Wachowski&quot;</td>
<td>&quot;V for Vendetta&quot;</td>
</tr>
<tr>
<td>&quot;Lana Wachowski&quot;</td>
<td>&quot;Speed Racer&quot;</td>
</tr>
<tr>
<td>&quot;Lilly Wachowski&quot;</td>
<td>&quot;Speed Racer&quot;</td>
</tr>
<tr>
<td>&quot;Nancy Meyers&quot;</td>
<td>&quot;Something's Gotta Give&quot;</td>
</tr>
</tbody>
</table>

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
```
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

<table>
<thead>
<tr>
<th>name</th>
<th>yearBorn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Lana Wachowski&quot;</td>
<td>1965</td>
</tr>
<tr>
<td>&quot;Jay Mohr&quot;</td>
<td>1970</td>
</tr>
<tr>
<td>&quot;River Phoenix&quot;</td>
<td>1970</td>
</tr>
<tr>
<td>&quot;Ethan Hawke&quot;</td>
<td>1970</td>
</tr>
<tr>
<td>&quot;Brooke Langton&quot;</td>
<td>1970</td>
</tr>
<tr>
<td>&quot;Tom Tykwer&quot;</td>
<td>1965</td>
</tr>
<tr>
<td>&quot;John C. Reilly&quot;</td>
<td>1966</td>
</tr>
</tbody>
</table>

Started streaming 7 records after 1 ms and completed after 2 ms.
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>
<thead>
<tr>
<th>p.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Keanu Reeves&quot;</td>
</tr>
</tbody>
</table>
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`
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.

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

<table>
<thead>
<tr>
<th>a.name</th>
<th>d.name</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Leni Riefenstahl&quot;</td>
<td>&quot;Leni Riefenstahl&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Emile Hirsch&quot;</td>
<td>&quot;Emile Hirsch&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Val Kilmer&quot;</td>
<td>&quot;Val Kilmer&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Gene Hackman&quot;</td>
<td>&quot;Gene Hackman&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Rich Yune&quot;</td>
<td>&quot;Rich Yune&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Audrey Tautou&quot;</td>
<td>&quot;Audrey Tautou&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Halle Berry&quot;</td>
<td>&quot;Halle Berry&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Cuba Gooding Jr.&quot;</td>
<td>&quot;Cuba Gooding Jr.&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Kevin Bacon&quot;</td>
<td>&quot;Kevin Bacon&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Tom Hardy&quot;</td>
<td>&quot;Tom Hardy&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Lawrence Fishburne&quot;</td>
<td>&quot;Lawrence Fishburne&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Hugo Weaving&quot;</td>
<td>&quot;Hugo Weaving&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Jay Maisie&quot;</td>
<td>&quot;Jay Maisie&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Hugo Weaving&quot;</td>
<td>&quot;Hugo Weaving&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Philip Seymour Hoffman&quot;</td>
<td>&quot;Philip Seymour Hoffman&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Werner Herzog&quot;</td>
<td>&quot;Werner Herzog&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

Started streaming 175 records after 8 ms and completed after 8 ms.
Collecting results

Find the movies that Tom Cruise acted in and return (aggregate) 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 and collecting results

Find all of the actors and directors who worked on a movie, return the **count** of the number of 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
```

Both **count** and **collect** are aggregate functions, so this is essentially a **group-by** query on `actor.name` and `director.name`
Chaining using WITH

Collect the movies that someone has appeared in, but show results only for people who’ve appeared in at least a half a dozen movies:

MATCH (person:Person)-[:ACTED_IN]->(movie:Movie)
WITH person, count(*) AS appearances, collect(movie.title) AS movies
WHERE appearances >= 6
RETURN person.name, appearances, movies

<table>
<thead>
<tr>
<th>person.name</th>
<th>appearances</th>
<th>movies</th>
</tr>
</thead>
</table>

Only columns declared in the WITH clause are visible after this point in the query.

No need for HAVING!
Updates: 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

122D Q: This an interesting graph DB feature – what are the implications for handling E-R models with inheritance...?

122D Q: What else might we say about “types” and schemas in the graph DB world?
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:

```cypher
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)
```

Removed 1 label, started streaming 1 records after 22 ms and completed after 22 ms.
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
```

Related to UPSERT!
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

This *is* UPSERT!
Adding or updating properties for a node - JSON style

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

```plaintext
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:

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

```sql
MATCH (a:Person)-[rel]-+(m:Movie) WHERE m.title = 'Batman Begins' RETURN a, rel, m
```

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

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

**Error:**

```
```

```
Neo.ClientError.Schema.ConstraintValidationFailed: Cannot delete node<1899>, because it still has relationships. To delete this node, you must first delete its relationships.
```

```
⚠️ Neo.ClientError.Schema.ConstraintValidationFailed: Cannot delete node<1899>, because it still has relationships. To delete this node, you must first...
```
Deleting a relationship and a node - 3

**Batman Begins** relationships:

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

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

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

Resulting *Michael Caine* nodes:

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

```mermaid
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
Let’s try some clarifying MERGE examples

```
MERGE (a:Actor {name: 'Michael Carey'})
SET a.born=1957
RETURN a
```

```
MERGE (a:Actor {name: 'Mike Carey'})
SET a.born=1957
RETURN a
```

```
MERGE (a:Actor {name: 'Mike Carey'})
SET a.born=1957, a.age=64
RETURN a
```

```
MATCH (p:Actor) WHERE p.name ENDS WITH 'Carey' RETURN p
```

**Important**: Only specify properties that will have unique keys when you merge.
Questions?
APPENDIX: Setting Up A Neo4J Query Playground
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
Setting up a development environment

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.

(Setup procedure video: https://youtu.be/8yWhuUnPapw)

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.
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
Exercise 1: Retrieving Nodes

In Neo4j Browser:

:play intro-exercises

Then follow instructions for Exercise 1.

(Proceed similarly for other exercises)