



Neo4j: A graph database

Algorithms, Data and Security
A.Y. 2024/25

Valeria Cardellini

Global Governance, 3rd year
Science and Technology Major

What is a database?

- Organized collection of structured information, or data, stored in a computer system
- Usually controlled by a **database management system (DBMS)**
- Together, data and DBMS, are referred to as a *database system*, often shortened to just database
- Data stored in a database can be easily accessed, managed, modified, updated, controlled, and organized

Types of databases

- Many types of databases, that mainly differ on data models
 - E.g., relational databases, NoSQL databases, graph databases
- The best type depends on how you intend to use data
- **Relational** databases
 - The most common type
 - Data is modeled in rows and columns in a series of tables to make processing and data querying efficient
 - Use **Structured Query Language (SQL)** for writing and querying data

Valeria Cardellini - ADS 2024/25

2

Neo4j: a graph database

- Graph database: database designed to treat relationships between data as equally important to (or even more important than) data itself
 - Purpose-built to store and navigate relationships
 - Uses nodes to store data entities, and links to store relationships between entities
- Neo4j: an open-source, native **graph database** <https://neo4j.com/>



Valeria Cardellini - ADS 2024/25

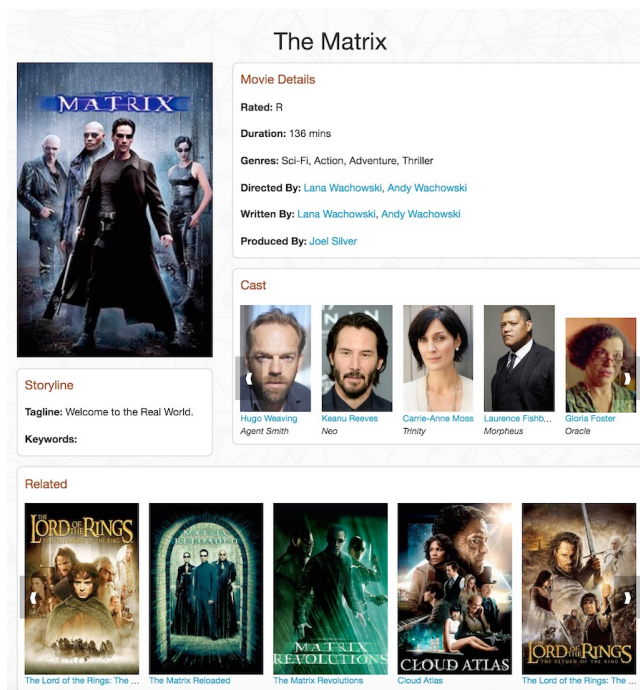
3

Graph data model

- Powerful data model
 - Designed to treat **relationships** between data
 - Focus on **visual representation** of information (more human-friendly)
- Data model based on **graph (network)** structure
 - *Nodes* are the **entities** and have a set of attributes
 - *Links* are the **relationships** between the entities
 - E.g.: an author writes a book

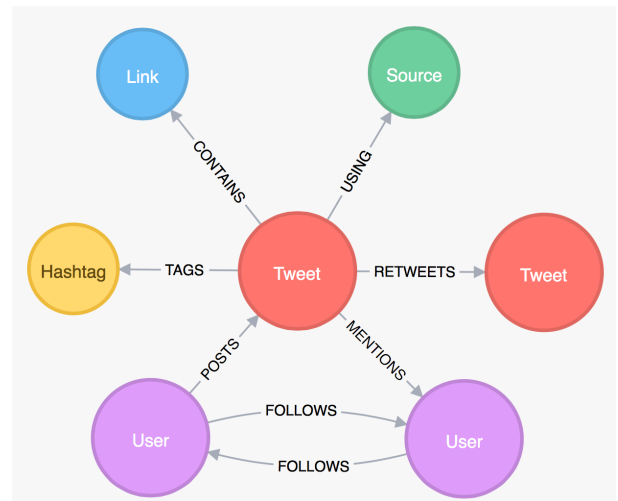
Graph data model: movies example

- How can we model information regarding the movie The Matrix?



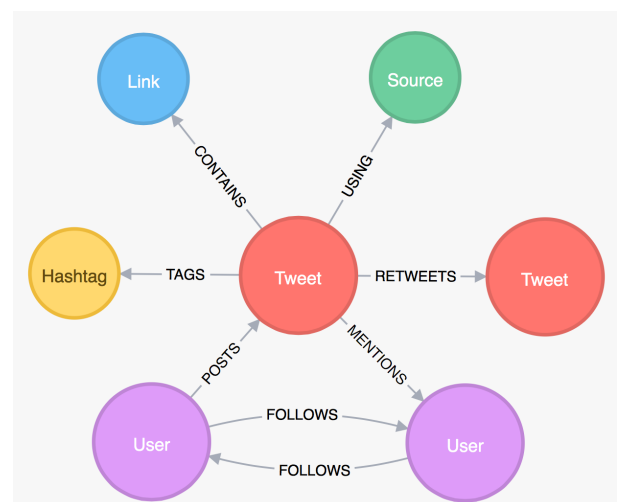
Graph data model: Twitter example

- How can we represent Twitter data and relationships?
- Choice depends also on what we want to analyze: let's assume social media activity
- **Nodes** in the graph model
 - User: represents a Twitter user
 - Tweet: represents a tweet
 - Hashtag: represents a hashtag
 - Link: represents a shared link in a tweet
 - Source: represents the platform used by Twitter users to tweet from



Graph data model: Twitter example

- **Relationships** in the graph:
 - POST relationship between a User and a Tweet: indicates that this user is the tweet author
 - RETWEETS relationship between two Tweets: indicates the first Tweet retweets the second Tweet
 - TAGS relationship between a Tweet and a Hashtag
 - FOLLOWS relationship between two Users: indicates the first User follows the second User
 - MENTIONS relationship between Tweet and User: indicates that the Tweet mentions the User

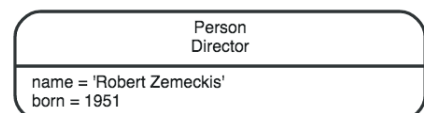
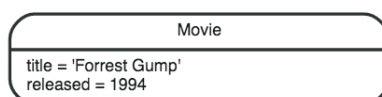
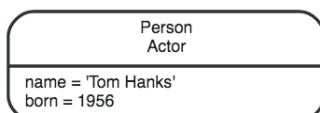


Suitable use cases for graph databases

- Good for applications where:
 - you need to model entities and relationships between them
 - and the focus is on querying for relationships between entities and analyzing relationships
- Some example of applications
 - Social network analysis
 - Recommendations
 - Fraud detection
 - Supply chain management

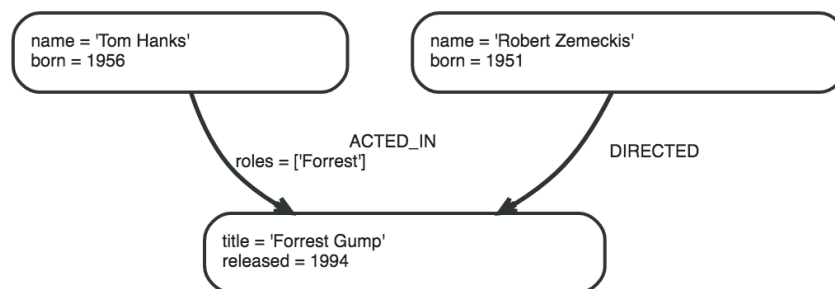
Neo4j: concepts

- Graph
 - Nodes, relationships, properties, and labels
- We use **nodes** to represent entities
 - A node can have properties and labels
 - A node can have relationships to other nodes, including itself
- Nodes and relationships have individual attributes called **properties**
- Properties consist of **key-value pairs**, e.g.,
 - name = 'Tom Hanks', born = 1956
 - title = 'Forrest Gump', released = 1994



Neo4j: concepts

- Nodes can be tagged with **labels** (i.e., node types)
 - Labels are used to group nodes into sets, so that all nodes with a given label belong to the same set (e.g., Actor and Director are labels for Person nodes)
- **Relationships** connect nodes, are unidirectional and can have properties
 - E.g., ACTED_IN, DIRECTED
- Properties are key-value pairs that are used to add qualities to nodes and relationships



Valeria Cardellini - ADS 2024/25

10

Neo4j: Cypher

- **Cypher**: Neo4j's graph query language
- Allows users to read data from and write data to Neo4j <https://neo4j.com/docs/getting-started/current/cypher-intro/>
- It uses a **declarative way** to query the graph powered by traversals and other techniques
 - A **traversal** navigates through the graph to find paths
 - Starts from starting nodes to related nodes, finding answers to queries
 - A **path** is one or more nodes with connecting relationships, typically retrieved as a query or traversal result
- It is a **textual** declarative query language
 - Uses a form of ASCII art to represent graph-related patterns
 - E.g., (:nodes) -[:ARE_CONNECTED_TO] -> (:otherNodes)

Valeria Cardellini - ADS 2024/25

11

Cypher syntax: node

- Cypher uses a pair of parentheses (), usually containing a text string, to represent a **node**

```
(varname:Label { p_name: p_value, ... } )
```

- () represents a node
- varname (optional) is a variable that we can assign to the node and use later in a query to refer to that node
- Label (prefixed with a colon :) declares node's type (or *label*)
- Node's properties are represented as a list of key/value pairs, enclosed within a pair of { }
- E.g., to represent a Person node with name and year of birth

```
(keanu:Person {name:'Keanu Reeves', born:1964})
```

variable

node type

properties as
key:'value' pairs

Cypher syntax: relationship

- Cypher uses a pair of dashes -- to represent an undirected **relationship**. Directed relationships have an arrowhead at one end <-- -->
- It is possible to create only directed relationships, although they can be queried as undirected
- Bracketed expressions [] are used to add details to a relationship:
 - We can assign a variable (e.g., role) also to a relationship and use it later in a query
 - Relationship's type (e.g., :ACTED_IN) is analogous to the node's label
 - Relationship's properties (e.g., roles) are analogous to node's properties

```
(keanu)-[role:ACTED_IN {roles:['Neo']}]>(TheMatrix)
```

variable

relationship type

properties as
key:'value' pairs

Cypher syntax: pattern variables

- To increase modularity and reduce repetition, Cypher allows **patterns to be assigned to variables**
 - This allows the matching paths to be inspected, used in other expressions, etc.
- E.g., `acted_in` is a variable

```
acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

Operations on data in Cypher

- We consider only a subset of operations available in Cypher
- Creating nodes and relationships and deleting them (write operations)
 - CREATE, DELETE
- Getting information on the graph model
- Querying the graph (read operations)
 - MATCH, RETURN
 - MATCH and WHERE
- Pattern matching and path finding

Cypher: CREATE

- Use **CREATE** to insert data (nodes and relationships) in the database
 - Example: create a node with label Person and property name with value John Doe
 - **RETURN** defines what to include in the query result

```
CREATE (p:Person {name: 'John Doe'})  
RETURN p
```



Cypher: CREATE

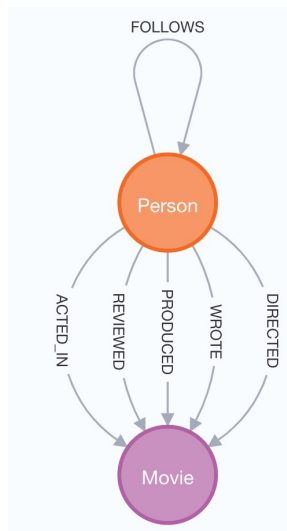
- Use **CREATE** to insert data (nodes and relationships)
 - Example: create a Person node and a Movie node and their relationship

```
CREATE (a:Person {name: 'Tom Hanks', born: 1956})-  
[r:ACTED_IN {roles: ['Forrest']}]>(m:Movie {title:  
'Forrest Gump', released: 1994})  
CREATE (d:Person {name: 'Robert Zemeckis', born:  
1951})-[:DIRECTED]>(m)  
RETURN a, d, r, m
```

Cypher: display graph model

- Once we have created the data (or used a pre-populated database), we can display the graph model in terms of node types and relationship types

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



Cypher: more information on graph

- To learn how many nodes there are in the graph, use

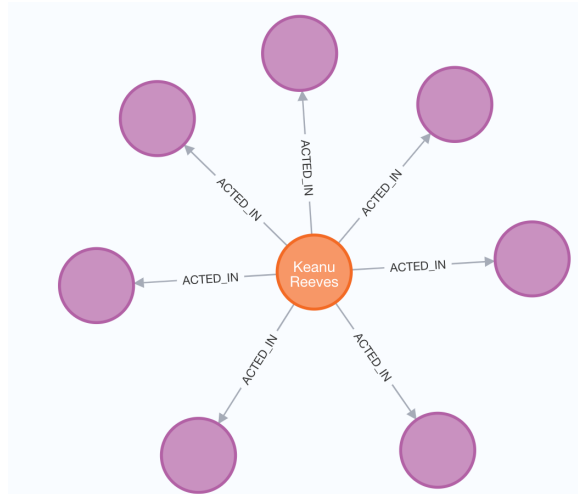
```
CALL apoc.meta.stats() YIELD labels
```

| | labels |
|---|---|
| 1 | { "Movie": 38, "Person": 133 } |

Cypher: MATCH

- Use **MATCH** to read data from database
 - **MATCH** specifies the patterns to search for in the database
 - E.g., find which movies Keanu Reeves has acted in

```
MATCH (keanu {name:'Keanu Reeves'})-[:ACTED_IN]->(movies:Movie) RETURN keanu, movies
```



Cypher: MATCH

- Use **MATCH** to read data from database
 - E.g., find which movies Keanu Reeves has acted in but now return only the movies title

```
MATCH (keanu {name:'Keanu Reeves'})-[:ACTED_IN]->(movies:Movie) RETURN movies.title
```

```
neo4j$ MATCH (keanu {name:'Keanu Reeves'})-[:ACTED_IN]->(movies:Movie) RETURN movies.title
```

| | movies.title |
|---|--------------------------|
| 1 | "The Matrix" |
| 2 | "The Matrix Reloaded" |
| 3 | "The Matrix Revolutions" |
| 4 | "The Devil's Advocate" |
| 5 | "The Replacements" |
| 6 | "Johnny Mnemonic" |
| 7 | "Something's Gotta Give" |

Cypher: MATCH and WHERE

- Use **WHERE** to add constraints to the patterns in a MATCH clause
 - E.g., find the movie with title The Matrix

```
MATCH (m:Movie)
WHERE m.title = 'The Matrix'
RETURN m
```

Cypher: DELETE

- Use **DELETE** to delete a node, e.g.,

```
MATCH (p:Person {name: 'John Doe'})
DELETE p
```

- Node cannot be deleted if it participates in a relationship. To remove also relationships, we need to detach the node, delete it and its relationships:

```
MATCH (d:Person {name: 'Greg Kinnear'})
DETACH DELETE d;
```

Cypher: search for patterns using length

- Cypher can be used to match patterns also of fixed, variable or unknown length
- Relationship pattern length:

```
(a) - [*2] -> (b)
```

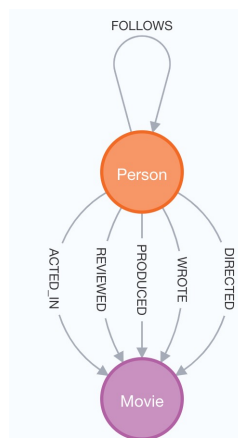
- It is possible to specify a length (e.g., 2) in the relationship description of a pattern

It can be a variable length:

*3..5 (between 3 and 5)
*3.. (greater than 3)
*..5 (less than 5)
* (any length)

Example: movie database

- Let's use as case study the movie database provided by Neo4j as sandbox with pre-populated data
 - Basic dataset of *Actors* acting in *Movies*
 - Available at <https://neo4j.com/sandbox/>
 - Data model of the movie database is



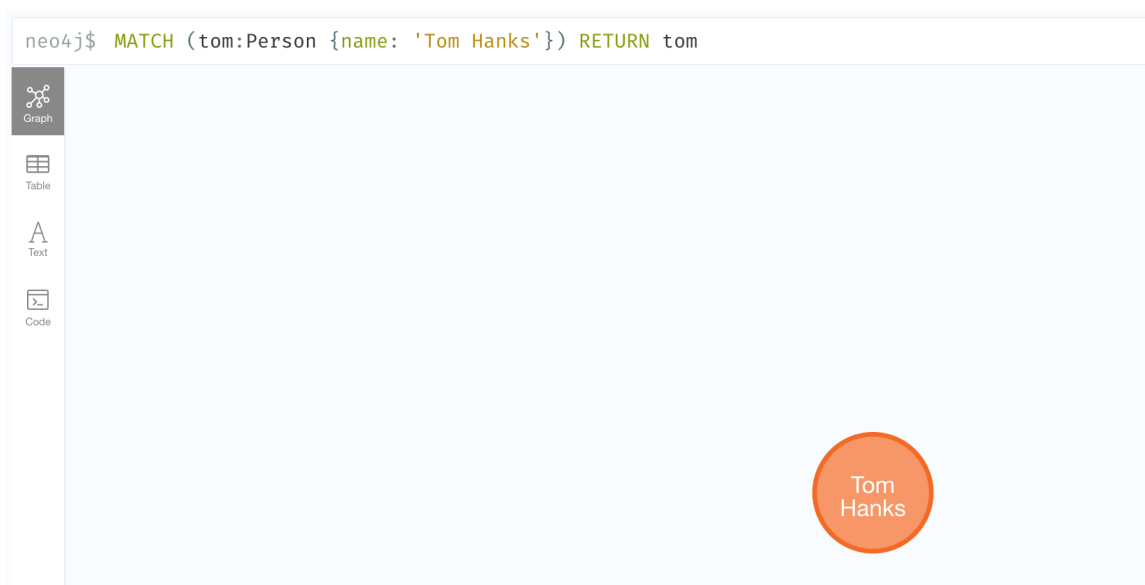
Example: movie database

- The goal of our analysis is to show recommendations for other actors to work with
 - By following the meaningful relationships between actors and movies, we can determine:
 - Occurrences of actors working together
 - Frequency of actors working with one another
 - Movies they have in common in the graph
- Let's start with simple queries and then increase their complexity

Some basic queries

- Let's find a single actor like *Tom Hanks*

`MATCH (tom:Person {name: 'Tom Hanks'}) RETURN tom`



Some basic queries

- Let's retrieve all Tom Hanks' movies by starting from Tom Hanks node and following ACTED_IN relationships

```
MATCH (tom:Person {name: 'Tom Hanks'})-[r:ACTED_IN]->(movie:Movie) RETURN tom, r, movie
```

- The query result looks like a graph



Valeria Cardellini - ADS 2024/25

28

Some basic queries

- Tom Hanks has colleagues who acted with him in his movies, let's find these **co-actors**:



```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coActor:Person) RETURN coActor.name
```

```
neo4j$ MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coActor:Person) RETURN coActor.name
```

| | coActor.name |
|---|------------------|
| 1 | "Parker Posey" |
| 2 | "Meg Ryan" |
| 3 | "Steve Zahn" |
| 4 | "Dave Chappelle" |
| 5 | "Ed Harris" |
| 6 | "Kevin Bacon" |
| 7 | |

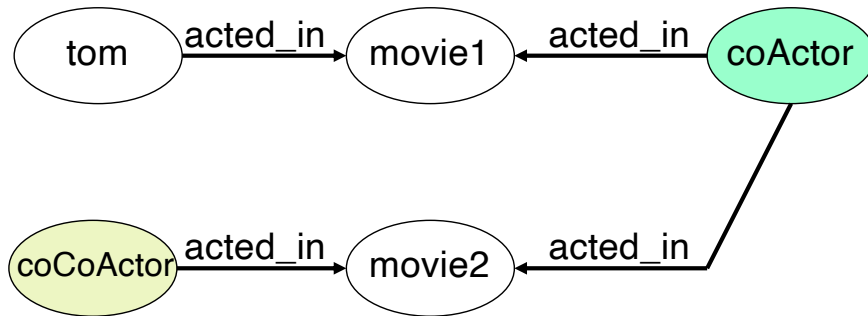
Started streaming 38 records after 2 ms and completed after 5 ms.

Valeria Cardellini - ADS 2024/25

29

Recommendations queries

- Let's find Tom's **co-co-actors**, i.e., the second-degree actors in Tom's network



Recommendations queries

- Let's find the **co-co-actors**, i.e., the second-degree actors in Tom's network. This will show us all the actors Tom may not have worked with yet, and we can specify a criterium to be sure he hasn't directly acted with that person

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(coCoActor:Person) WHERE tom <> coCoActor AND NOT (tom)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coCoActor) RETURN coCoActor.name
```


Recommendations queries

- In the query result a few names appear multiple times, because there are multiple paths to follow from *Tom Hanks* to these actors

```
neo4j$ MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)-[:ACTED_IN]-(coActor:Person)-[:ACTED_IN]->(movie2:M...
```

| | coCoActor.name |
|---|-------------------|
| 1 | "Val Kilmer" |
| 2 | "Tom Skerritt" |
| 3 | "Kelly McGillis" |
| 4 | "Tom Cruise" |
| 5 | "Anthony Edwards" |
| 6 | "Carrie Fisher" |
| 7 | "Billy Crystal" |
| 8 | "Bruno Kirby" |
| 9 | "Zach Grenier" |

Recommendations queries

- Let's see which co-co-actors appear most often in Tom's network: we can take frequency of occurrences into account by counting the number of paths between *Tom Hanks* and each coCoActor and ordering them by highest to lowest value

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)-[:ACTED_IN]-(coActor:Person)-[:ACTED_IN]->(movie2:Movie)-[:ACTED_IN]-(coCoActor:Person) WHERE tom <> coCoActor AND NOT (tom)-[:ACTED_IN]->(movie1)-[:ACTED_IN]-(coCoActor) RETURN coCoActor.name, count(coCoActor) as frequency ORDER BY frequency DESC LIMIT 5
```

Recommendations queries

- The query result

```
neo4j$ MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]-(movie1:Movie)-[:ACTED_IN]-(coActor:Person)-[:ACTED_IN]-(movie2:Movie)-[:ACTED_IN]-(coCoActor:Person) WHERE tom <> coCoActor AND NOT (tom)-[:ACTED_IN]-(movie)-[:ACTED_IN]-(coCoActor) RETURN coCoActor.name, count(coCoActor) as frequency ORDER BY frequency DESC LIMIT 5
```

| | coCoActor.name | frequency |
|---|-------------------|-----------|
| 1 | "Tom Cruise" | 5 |
| 2 | "Zach Grenier" | 5 |
| 3 | "Keanu Reeves" | 4 |
| 4 | "Kelly McGillis" | 3 |
| 5 | "Anthony Edwards" | 3 |

Recommendations queries

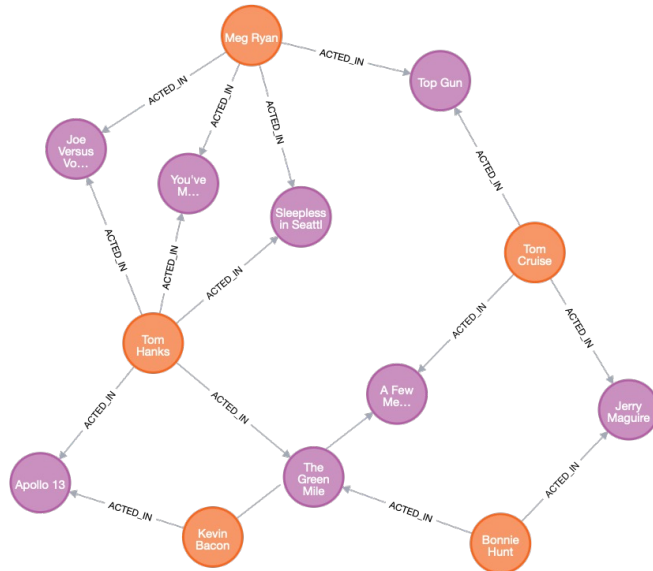
- One of the most frequent “co-co-actors” is *Tom Cruise*. Now let’s see which movies and actors are between the two Toms so we can find out who can introduce them

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)-[:ACTED_IN]-(coActor:Person)-[:ACTED_IN]->(movie2:Movie)-[:ACTED_IN]-(cruise:Person {name: 'Tom Cruise'}) WHERE NOT (tom)-[:ACTED_IN]->(:Movie)-[:ACTED_IN]-(cruise) RETURN tom, movie1, coActor, movie2, cruise
```

Recommendations queries

- The query result: there are multiple paths between the two Toms
 - And there is Kevin Bacon in one of the paths: see the six degrees of Kevin Bacon game

https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon



Valeria Cardellini - ADS 2024/25

36

Degree centrality

- The Neo4j [Graph Data Science \(GDS\)](https://neo4j.com/docs/graph-data-science/current/management-ops/graph-creation/graph-project/) library contains many graph algorithms
- Let's find which actor has acted in the most movies using degree centrality
- We first need to create a graph projection using **`gds.graph.project`** procedure

<https://neo4j.com/docs/graph-data-science/current/management-ops/graph-creation/graph-project/>

```
CALL gds.graph.project(
  'proj',
  ['Person', 'Movie'],
  'ACTED_IN'
);
```

Valeria Cardellini - ADS 2024/25

37

Degree centrality

- Then we run the degree centrality algorithm on the projected graph using `gds.degree.stream` procedure
- We also order the results to determine which actor has directed the most movies

```
CALL gds.degree.stream('proj')
YIELD nodeId, score
RETURN
    gds.util.asNode(nodeId).name AS actorName,
    score AS numberOfMoviesActedIn
ORDER BY numberOfMoviesActedIn DESCENDING,
actorName LIMIT 5
```

Degree centrality

- The query result is

| | actorName | numberOfMoviesActedIn |
|---|------------------|-----------------------|
| 1 | "Tom Hanks" | 12.0 |
| 2 | "Keanu Reeves" | 7.0 |
| 3 | "Hugo Weaving" | 5.0 |
| 4 | "Jack Nicholson" | 5.0 |
| 5 | "Meg Ryan" | 5.0 |

Started streaming 5 records after 10 ms and completed after 124 ms.

Shortest path

- What is the shortest path between Kevin Bacon and Clint Eastwood?
- We first need to create a graph projection using **`gds.graph.project`** procedure

```
// Create projection
CALL gds.graph.project(
  'proj2',
  ['Person', 'Movie'],
  {
    ACTED_IN:{orientation:'UNDIRECTED'},
    DIRECTED:{orientation:'UNDIRECTED'}
  }
);
```

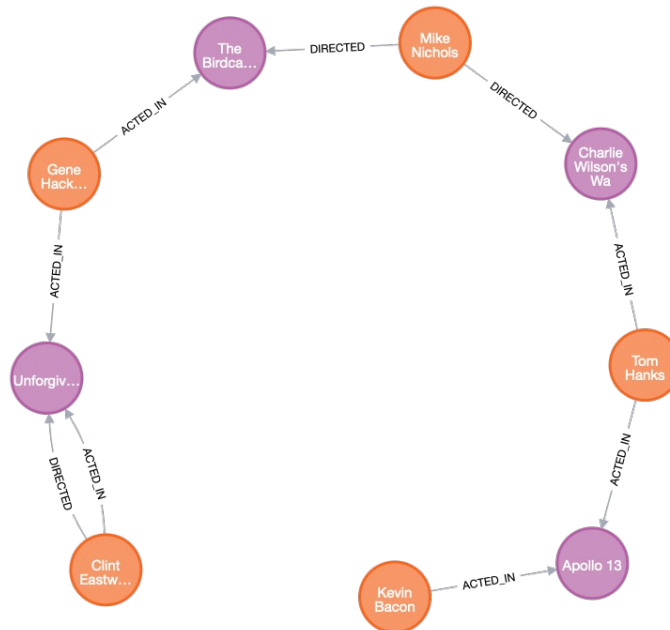
Shortest path

- We match the 2 Person nodes and then use **`gds.shortestPath.dijkstra.stream`** procedure to find the shortest path

```
MATCH (kevin:Person{name : 'Kevin Bacon'})
MATCH (clint:Person{name : 'Clint Eastwood'})
CALL gds.shortestPath.dijkstra.stream(
  'proj2',
  {
    sourceNode:kevin,
    TargetNode:clint
  }
)
YIELD sourceNode, targetNode, path
RETURN sourceNode, targetNode, nodes(path) as path;
```

Shortest path

- The query result is



Neo4j sandboxes

- As project for the course, you are going to use one of the Neo4j sandboxes
 - Online tool, not requiring a local installation
<https://neo4j.com/sandbox/>
 - Pre-populated with domain data and focus on use-case specific queries
 - See sandbox description on the course web site
 - Each sandbox is available for 3 days after creation and can be extended for 7 additional days before expiration; after the additional days, you need to restart the sandbox from scratch (in this case, you will lose new data you have written to the sandbox)

References

- Neo4J fundamentals (1-hour course)
<https://graphacademy.neo4j.com/courses/neo4j-fundamentals/>
- Cypher fundamentals (1-hour course)
<https://graphacademy.neo4j.com/courses/cypher-fundamentals/>