TOR VERGATA
UNIVERSITY OF ROME

Neo4j: A graph database

Algorithms, Data and Security
A.Y. 2025/26

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

Neo4j: a graph database

- Graph database: a database designed to treat relationships between data as equally important to as the 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/

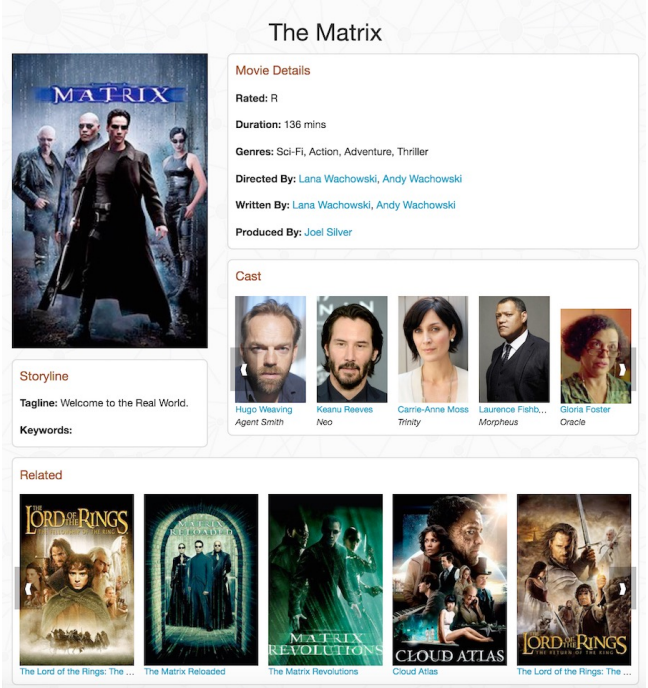
Graph data model

- 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, each with have a set of attributes
    - E.g.: author, book
  - *Links* are the relationships between the entities
    - E.g.: an author writes a book

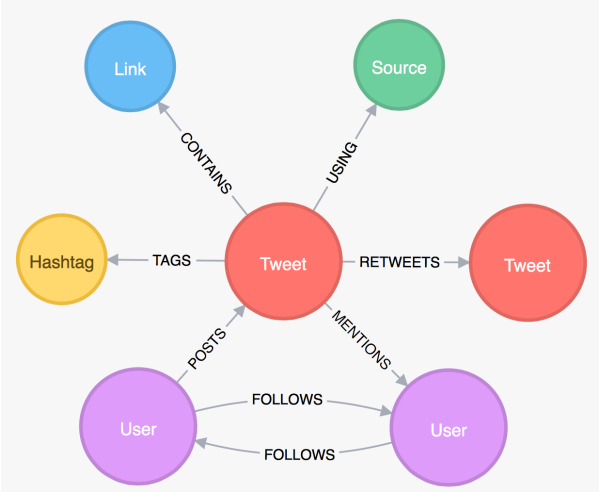
Graph data model: movie example

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



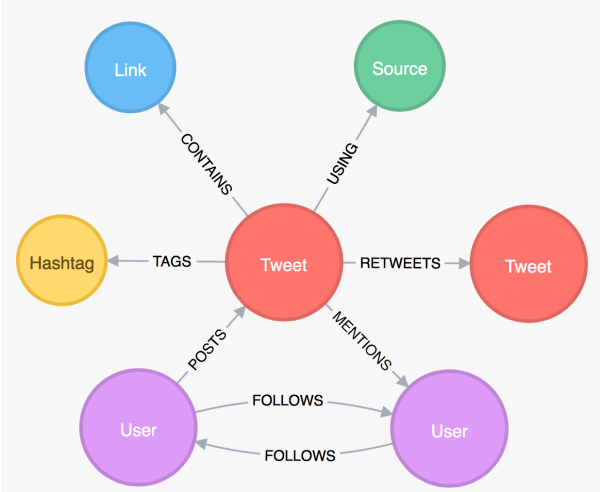
Graph data model: Twitter example

- How can we represent Twitter data and relationships?
- The choice depends on the type of analysis: we assume social media activity
- Nodes (entities)
  - `User`: represents a Twitter user
  - `Tweet`: represents a tweet
  - `Hashtag`: represents a hashtag used in tweet
  - `Link`: represents a URL shared in a tweet
  - `Source`: represents the platform used to publish the tweet (e.g., mobile, web)



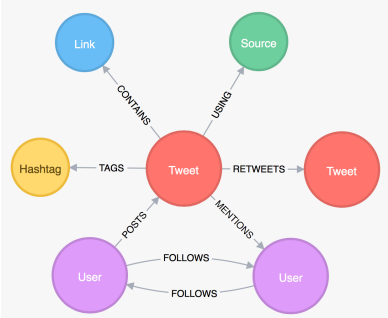
Graph data model: Twitter example

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



Graph data model: Twitter example

- Why this model is useful
  - Analyze influence (who follows whom)
  - Track trending topics via hashtags
  - Study content sharing (links, retweets)
  - Understand user behavior across platforms

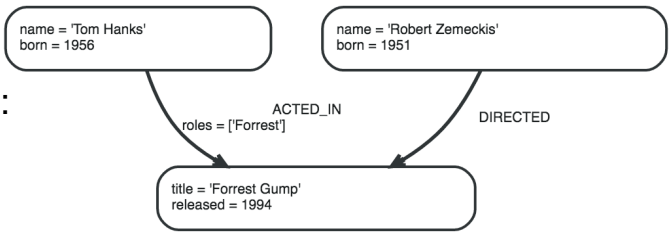


Suitable use cases for graph databases

- Ideal for applications where:
  - you need to model entities and relationships between them
  - and the focus is on querying and analyzing relationships
- When to use a graph database
  - Highly connected data
  - Complex relationships that are difficult to model with tables
  - Frequent relationship traversal (e.g., multi-hop queries)
- Example applications
  - Social network analysis
  - Recommendation systems
  - Fraud detection
  - Supply chain management

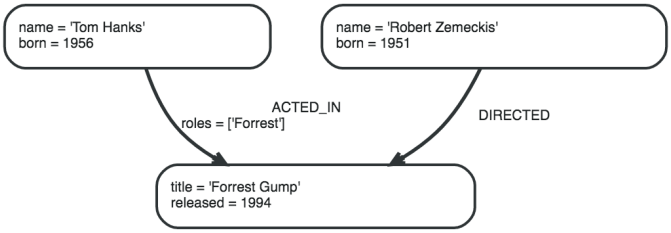
Neo4j: concepts

- A graph consists of
  - Nodes, relationships, properties, and labels
- Nodes represent entities
- Nodes can have:
  - Labels (to define their type, e.g., Person, Movie)
  - Properties (attributes as key-value pairs)
  - Relationships to other nodes, including themselves
- Relationships
  - Represent connections between nodes
  - Are directed and can have:
    - A type (e.g., ACTED_IN)
    - Properties



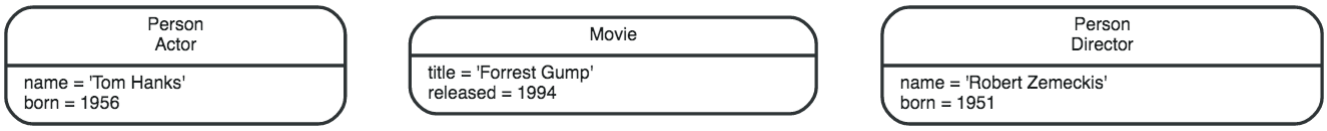
Neo4j: concepts

- Nodes and relationships have individual attributes called properties
- Properties
  - Key-value pairs describing nodes or relationships
- Examples:
  - Person node:
    - name = "Tom Hanks"
    - born = 1956
  - Movie node:
    - title = "Forrest Gump"
    - released = 1994



Neo4j: concepts

- Node labels
  - Nodes can be tagged with labels (i.e., node types)
  - Labels are used to group nodes into sets (e.g., Person)
  - All nodes with the same label belong to the same group



Neo4j: Cypher

- Cypher is Neo4j query language
- Used to read and write data in the database
  https://neo4j.com/docs/getting-started/current/cypher-intro/
- Features
  - Declarative language: specify what to retrieve, not how
  - Based on graph traversals and pattern matching
  - Uses intuitive, human-readable syntax
- Concepts
  - Traversal
    - Navigates the graph to find paths
    - Starts from one or more nodes and follows relationships
  - Path
    - A sequence of nodes connected by relationships
    - Returned as the result of a query

Neo4j: Cypher

- Syntax style
  - Textual and expressive
  - Uses ASCII-art patterns to represent graphs
- Example:

```
(:Node)-[:ARE_CONNECTED_TO]->(:OtherNode)
```


Cypher: node

- Nodes are represented using parentheses ( )
- Node structure

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

  - ( ) represents a node
  - `varname` (optional): variable used to refer that node in a query
  - `:Label` declares node's type (or *label*)
  - { } contains properties as key-value pairs
  - E.g., to represent a Person node with name and year of birth

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

variable

label

properties as `key:'value'` pairs

Cypher: relationship

- Relationships are represented using dashes:
  - `--` for undirected relationship
  - `-->` or `<--` for directed relationship
- In practice, relationships are stored as directed, but can be queried in both ways

Cypher: relationship

- Relationship structure

```
()-[var:TYPE {properties}]->()
```

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

variable — relationship type — properties as `key:'value'` pairs

- `[ ]` defines a relationship
- `var` (optional): variable name for the relationship (e.g., `role`)
- `:TYPE` relationship type (similar to node labels)
- `{ }` properties as key-value pairs

– We can assign a variable also to a relationship and use it later in a query

– Relationship's type (e.g., `:ACTED_IN`) is analogous to node's label

– Relationship's properties (e.g., `roles:['Neo']`) are analogous to node's properties

Cypher: pattern variables

- To increase modularity and reduce repetition, Cypher allows graph patterns to be assigned to variables
  - These variables can be reused throughout a query
  - This allows to:
    - Inspect matched paths
    - Reuse patterns in other expressions
    - Improve query readability
- E.g., `acted_in` is a variable

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

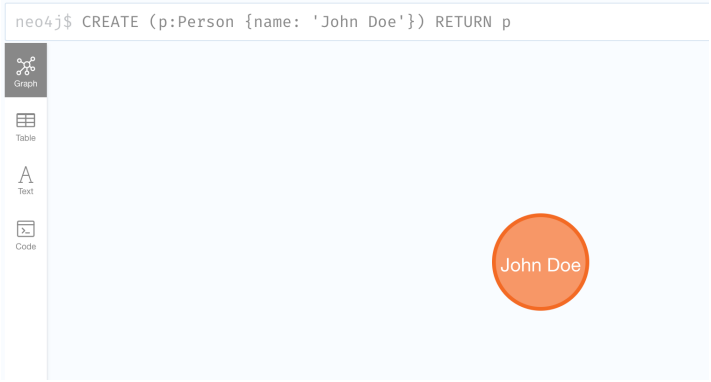

Cypher: operations

- We consider a subset of Cypher operations focused on:
- Write operations
  - Create nodes and relationships and delete them (write operations)
    - `CREATE`, `DELETE`
- Read operations
  - Retrieve information on graph model
  - Query the graph
    - `MATCH`, `RETURN`
    - `MATCH` and `WHERE`
  - Cypher queries are based on pattern matching and path traversal in the graph

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

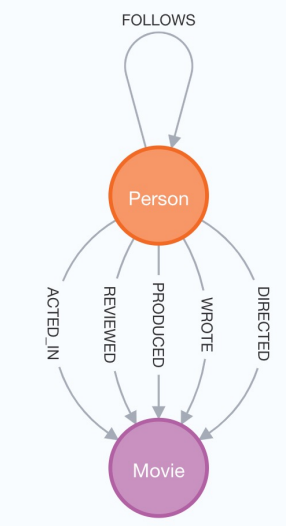
- Use `CREATE` to insert data (nodes and relationships)
  - Example: create a Person node and a Movie node and their relationship

```
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 data is created (or a pre-populated database is used), visualize the graph model in terms of nodes and relationships

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



Cypher: more information on graph

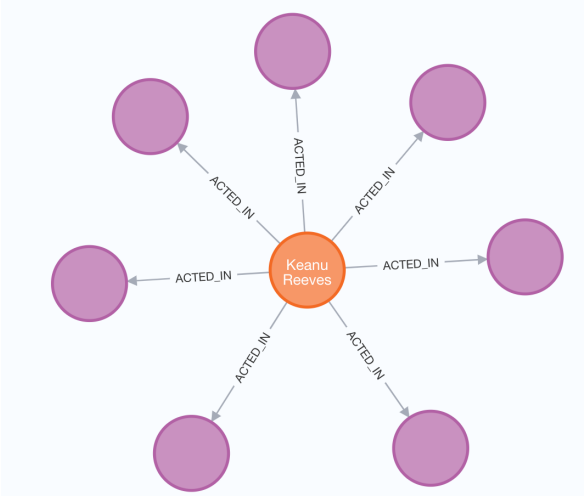
- To learn how many nodes there are in the graph, use

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

Cypher: MATCH

- Use `MATCH` to read data from the database
  - `MATCH` specifies patterns to search for in the graph
  - `RETURN` specifies what data to output

```
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, return only the movie titles (no the full movie nodes)

```
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 patterns in a `MATCH` clause
  - It filters the results based on conditions
  - 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
```

- A node cannot be deleted if it still participates in a relationship. To delete a node all its relationships, use `DETACH DELETE`:

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

Cypher: search for patterns using length

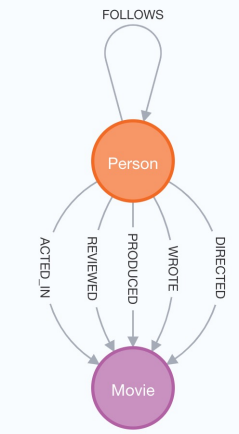
- Cypher can match graph patterns with fixed, variable or unknown length
- Relationship pattern length:

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

- You can specify the number of hops (e.g., 2) in a relationship pattern
- It can be a variable length:
  - `*3..5` (between 3 and 5)
  - `*3..` (3 or more)
  - `*..5` (up to 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



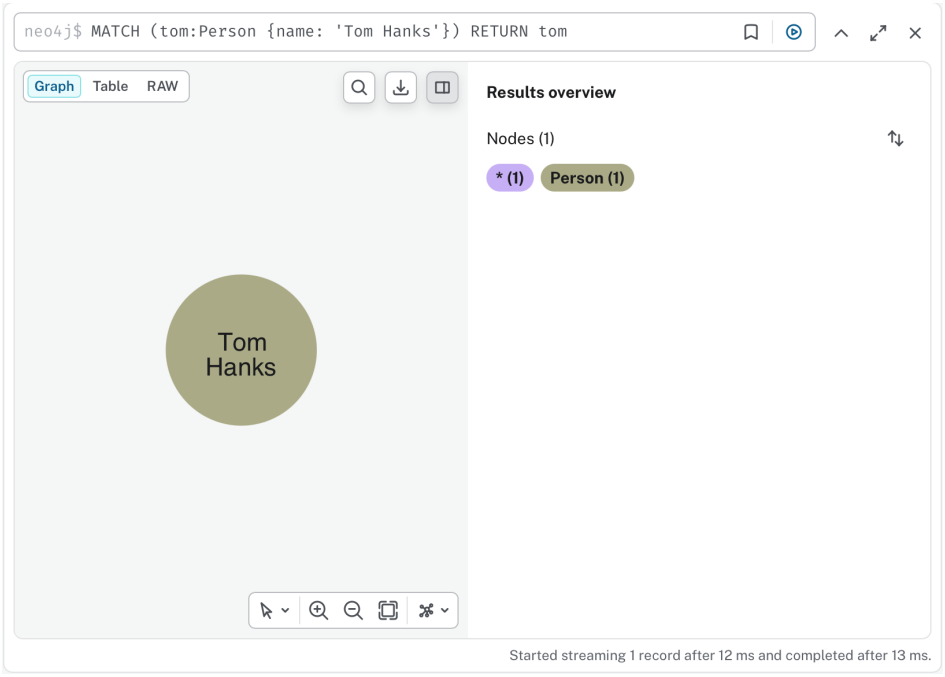
Example: movie database

- The goal of our analysis is to generate recommendations for actors to collaborate with other actors
  - By traversing meaningful relationships between actors and movies, we can determine:
    - Actors who have worked together
    - Frequency of collaborations between actors
    - Movies that actors have in common in the graph
- Start with simple queries and gradually increase their complexity

Basic queries

- Find a single actor like *Tom Hanks*

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



Basic queries

- Retrieve Tom Hanks' movies by traversing relationships

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

  - The result looks like a graph

Some basic queries

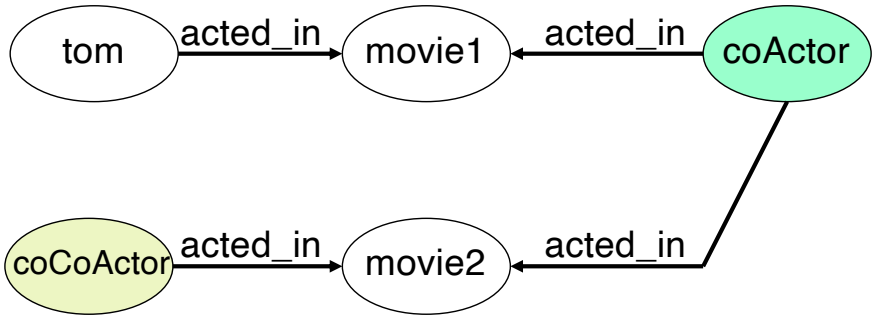
- Find co-actors, that is actors who acted with Tom Hanks in the same movies:


```
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)<-[:AC
Table RAW
coActor.name
1 "Ed Harris"
2 "Gary Sinise"
3 "Kevin Bacon"
4 "Bill Paxton"
5 "Parker Posey"
6 "Greg Kinnear"
7 "Meg Ryan"
8 "Steve Zahn"
9 "Dave Chappelle"
10 "Madonna"
```

Recommendations queries

- Find actors who worked with Tom Hanks' co-actors (co-co-actors that is, second-degree connections)



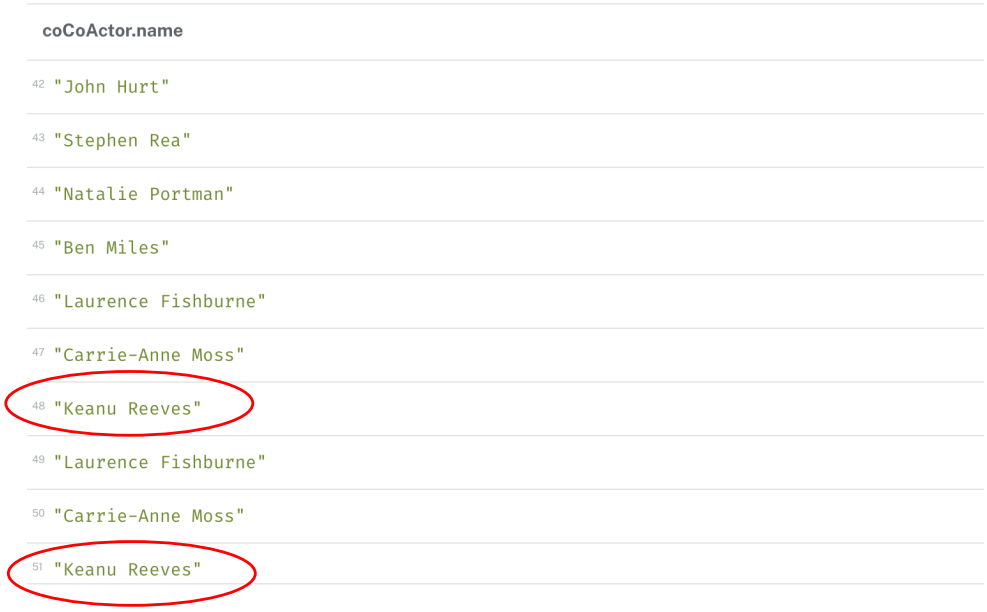
Recommendations queries

- Find Tom Hanks' 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 criterion to ensure he has not 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 some actor names names appear multiple times, because there are multiple paths from *Tom Hanks* to these actors in the graph



	coCoActor.name
42	"John Hurt"
43	"Stephen Rea"
44	"Natalie Portman"
45	"Ben Miles"
46	"Laurence Fishburne"
47	"Carrie-Anne Moss"
48	"Keanu Reeves"
49	"Laurence Fishburne"
50	"Carrie-Anne Moss"
51	"Keanu Reeves"

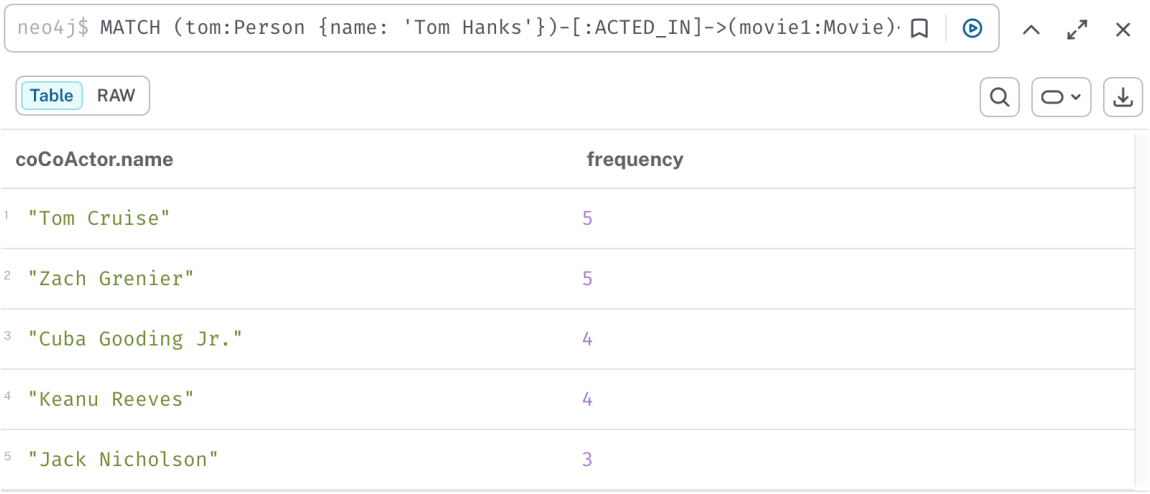
Recommendations queries

- 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]->(:Movie)<-[:ACTED_IN]-
(coCoActor) RETURN coCoActor.name, count(coCoActor)
as frequency ORDER BY frequency DESC LIMIT 5
```

Recommendations queries

- The query result



Recommendations queries

- One of the most frequent “co-co-actors” is *Tom Cruise*. Now we find which movies and actors are between the two Toms so we can discover 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)
WITH tom, movie1, coActor, movie2, cruise
MATCH path = (tom)-[:ACTED_IN]->(movie1)<-
[:ACTED_IN]-(coActor)-[:ACTED_IN]->(movie2)<-
[:ACTED_IN]-(cruise)
RETURN tom, movie1, coActor, movie2, cruise,
relationships(path) AS links;
```

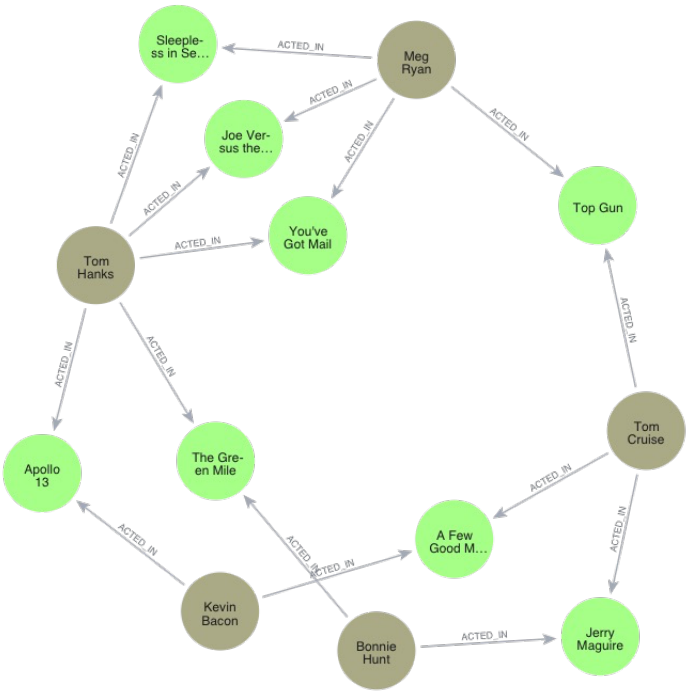
Recommendations queries

- The query result: there are multiple paths connecting the two Toms
  - One of the paths include Kevin Bacon: see the six degrees of Kevin Bacon game https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

	tom	movie1	coActor	movie2	cruise
1	Tom Hanks	The Green Mile	Bonnie Hunt	Jerry Maguire	Tom Cruise
2	Tom Hanks	Sleepless in Seattle	Meg Ryan	Top Gun	Tom Cruise
3	Tom Hanks	You've Got Mail	Meg Ryan	Top Gun	Tom Cruise
4	Tom Hanks	Joe Versus the Volcar	Meg Ryan	Top Gun	Tom Cruise
5	Tom Hanks	Apollo 13	Kevin Bacon	A Few Good Men	Tom Cruise

Recommendations queries

- The same result as graph



Degree centrality

- The Neo4j Graph Data Science (GDS) library provides many graph algorithms
  - Used for advanced analysis such as centrality, communities, and path finding
- Find the actor who has acted in the most movies using degree centrality
- Before running algorithms, we create an in-memory graph projection using `gds.graph.project`

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


Degree centrality

- Then we run degree centrality on the projected graph using `gds.degree.stream` procedure
- We also order the results to find the actor who has acted in the most movies

```
CALL gds.degree.stream('actorGraph')
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

```
neo4j$ CALL gds.degree.stream('proj') YIELD nodeId, score RETURN gds.util
```

Table RAW

	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

Shortest path

- Find the shortest path between Kevin Bacon and Clint Eastwood
- Before running the algorithm, we build an in-memory graph using the GDS library
  - We make relationships UNDIRECTED to allow traversal in both directions

```
CALL gds.graph.project(
    'actorGraph2',
    ['Person','Movie'],
    {
        ACTED_IN:{orientation:'UNDIRECTED'},
        DIRECTED:{orientation:'UNDIRECTED'}
    }
);
```

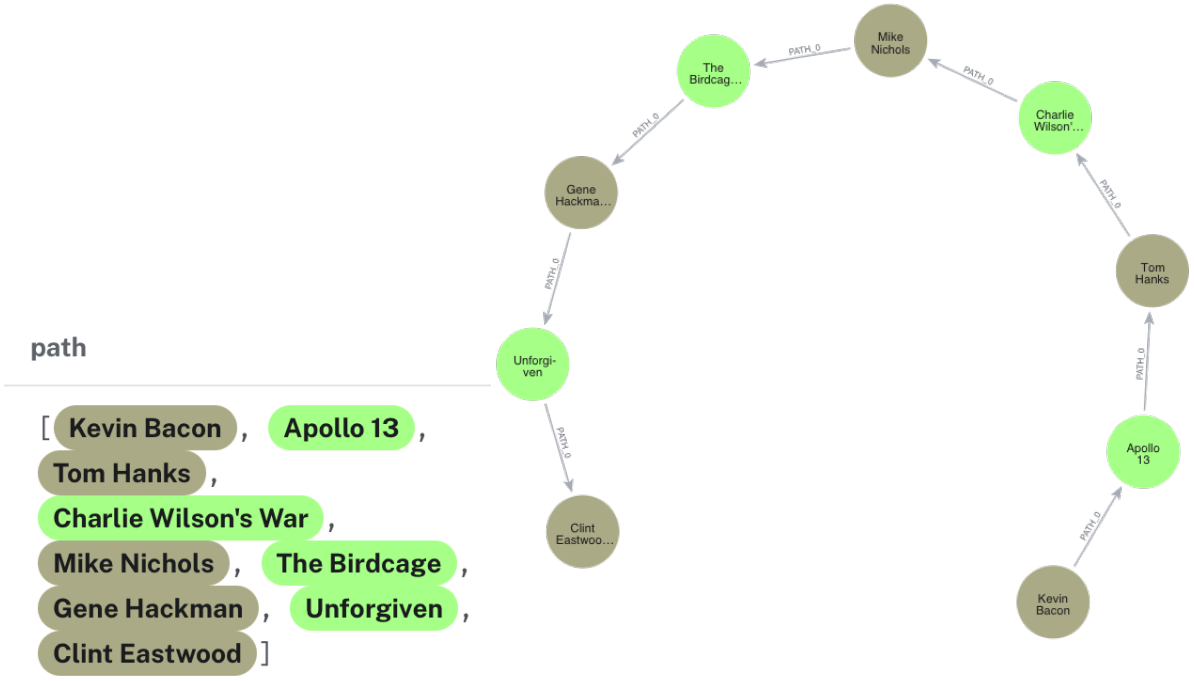
Shortest path

- We match the 2 Person nodes and then use Dijkstra algorithm to find the shortest path between them

```
MATCH (kevin:Person {name: 'Kevin Bacon'})
MATCH (clint:Person {name: 'Clint Eastwood'})
CALL gds.shortestPath.dijkstra.stream(
    'actorGraph2',
    {
        sourceNode: kevin,
        targetNode: clint
    }
)
YIELD sourceNode, targetNode, path
RETURN sourceNode, targetNode, nodes(path) AS path,
relationships(path) AS links;
```


Shortest path

- The query result is



Neo4j sandboxes

- As project for the course, you will use one of the Neo4j sandboxes
  - Online tool, not requiring a local installation https://neo4j.com/sandbox/
  - Pre-populated with domain data and designed for use-case specific queries
    - See sandbox description on the course website
  - Each sandbox is available for 3 days after creation and can be extended for 7 additional days before expiration; after that, you must restart the sandbox from scratch and any newly created data will be lost
- Based on the sandbox data, run queries, visualize the data, and analyze the results
- Try to run different queries from those already proposed

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/