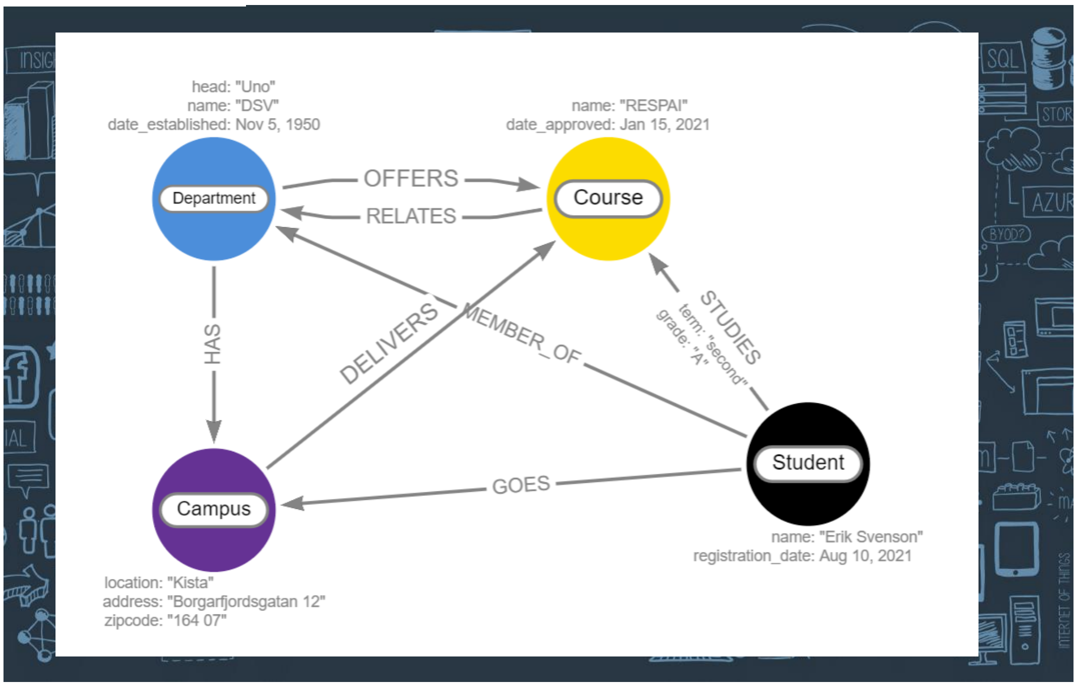
Stockholms universitet



Advanced querying and visualization of graph data

RESPAI

| Scalable and Responsible AI in Organizations | VT2024

Compiled by Workneh Yilma Ayelee

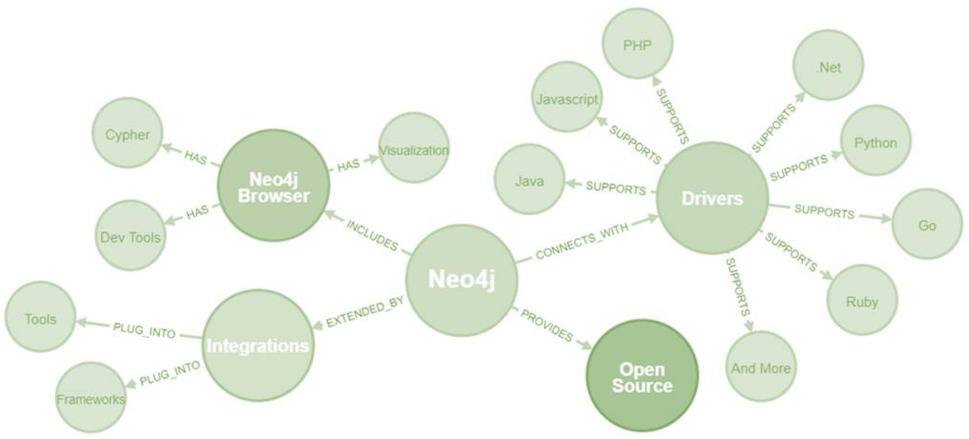


Table of Contents

Objectives of this tutorial

- Familiarize yourself with graph database
- Familiarize yourself with graph database basic concepts
- Learn about Neo4j Cypher to query graph data
- Learn about Neo4j Bloom for more enhanced visualization of graph data analytics

1. Querying and Building Bloom Visualization

In this section, you will learn basic querying of the Neo4j database and bloom visualization.

- Close your Neo4j browser if it is still open
- From the Neo4j Desktop, click on **Stop** GraphDBMS1 if it is still running
- Create a new project – *Intro_Query_Bloom*
- Under the *Intro_Query_Bloom* project, click on **Add**
- Create a new local database by clicking on **Local DBMS**
- Under **Name** enter *MyMovies*
- Under **Password** enter *graph1234*
- Under **Version** select 4.4.0
- Click on **Create**

Install APOC and Graph Data Science Library plugins

- After the database is created, click on it
- Select **Plugins**
- Expand **APOC** and click on **Install**
- Expand **Data Science Library** and click on **Install**

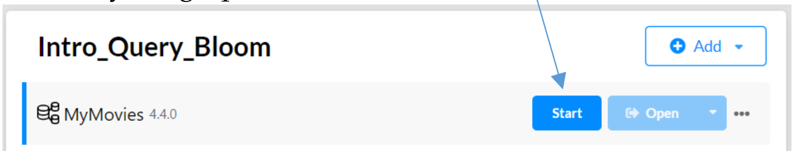
Now your Neo4j Desktop is ready

1.1. Querying graph database in Neo4j – Movies Database

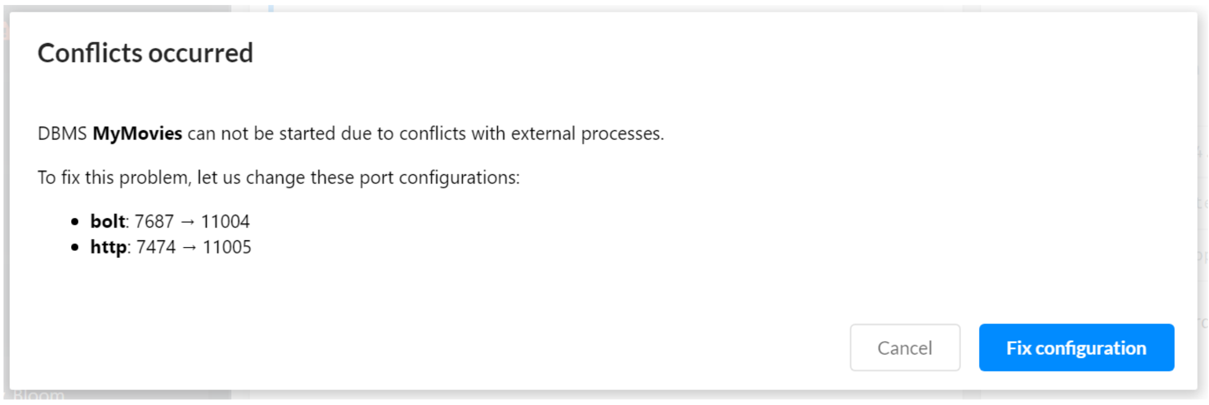
In this tutorial, you will use a sample Neo4j graph database Movies. The Movies graph contains nodes representing movies, actors, directors and their relations pertaining to collaboration for making real. The graph data is already created, i.e., the Cypher for creating the graph database is already available through the Cypher -:Play *movies*.

Start the graph database *MyMovies*

- To start your graph database click on **Start**.



- **Note:** You may find the following message: If so click on **Fix Configuration**!



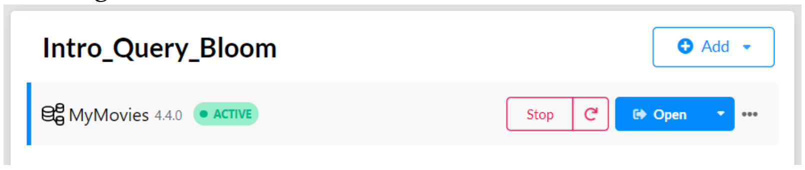
Conflicts occurred

DBMS **MyMovies** can not be started due to conflicts with external processes.

To fix this problem, let us change these port configurations:

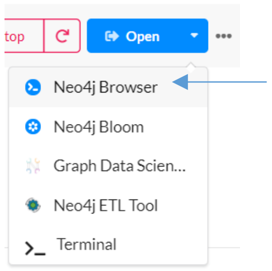
- **bolt**: 7687 → 11004
- **http**: 7474 → 11005

- Your graph database should be active to proceed with the next steps as illustrated in the diagram below.

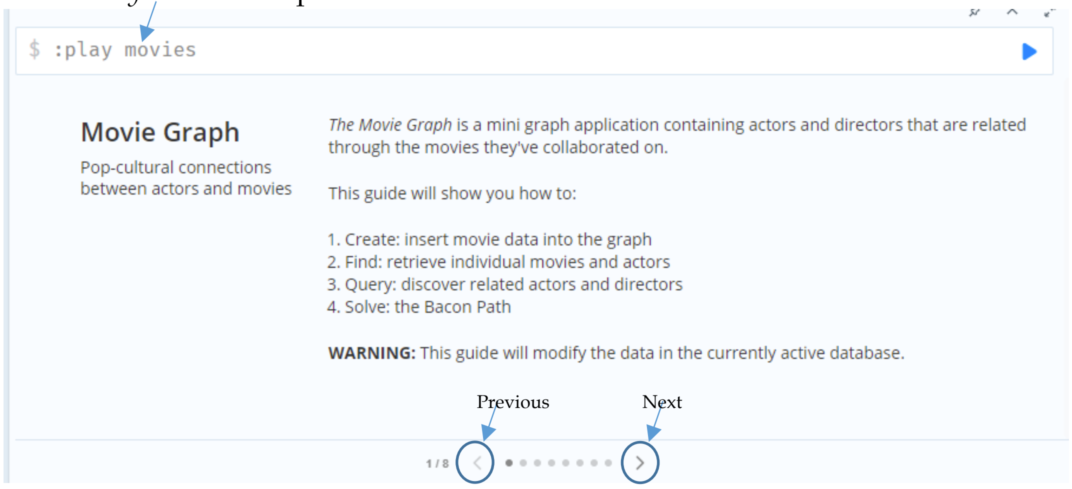


Start the Neo4j Browser

- Click on **Open** ,or if you click on the drop arrow of the **Open** button, select **Neo4j Browser**



- Enter *:Play movies* and press **Enter**



```
$ :play movies
```

Movie Graph

Pop-cultural connections between actors and movies

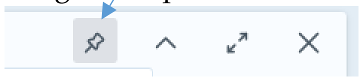
The Movie Graph is a mini graph application containing actors and directors that are related through the movies they've collaborated on.

This guide will show you how to:

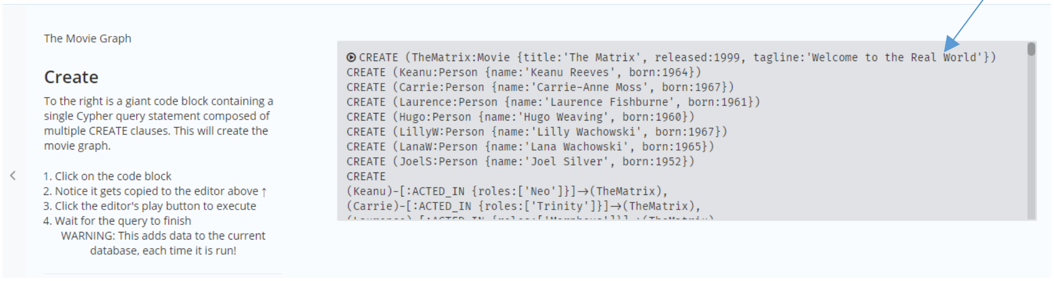
1. Create: insert movie data into the graph
2. Find: retrieve individual movies and actors
3. Query: discover related actors and directors
4. Solve: the Bacon Path

WARNING: This guide will modify the data in the currently active database.

- Click on **Pin at top** so that your step-through guide will remain on the top as you go along the steps.



- Click on **Next (to page2)**
- You will see a long Cypher query containing several CREATE clauses (with approximately 500 lines of codes) to enable you to create the movie graph > Click on the code, and the code will be added to the command line.



The Movie Graph

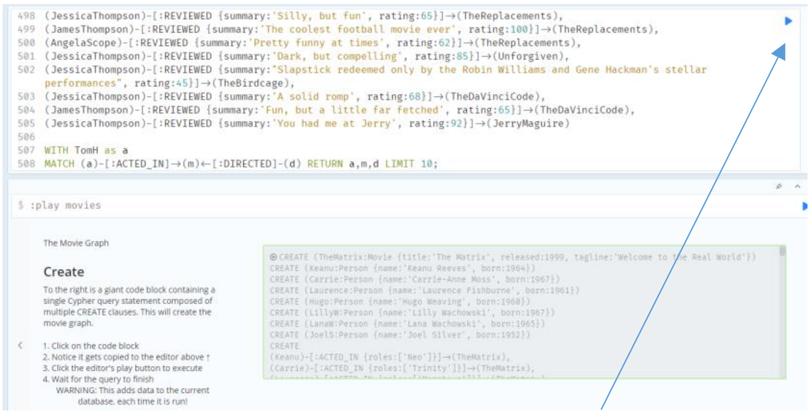
Create

To the right is a giant code block containing a single Cypher query statement composed of multiple CREATE clauses. This will create the movie graph.

1. Click on the code block
2. Notice it gets copied to the editor above ↑
3. Click the editor's play button to execute
4. Wait for the query to finish
   WARNING: This adds data to the current database, each time it is run!

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to the Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Carrie:Person {name:'Carrie-Anne Moss', born:1967})
CREATE (Laurence:Person {name:'Laurence Fishburne', born:1961})
CREATE (Hugo:Person {name:'Hugo Weaving', born:1960})
CREATE (LillyW:Person {name:'Lilly Wachowski', born:1967})
CREATE (LanaW:Person {name:'Lana Wachowski', born:1965})
CREATE (JoelS:Person {name:'Joel Silver', born:1952})
CREATE
(Keanu)-[:ACTED_IN {roles:['Neo']}]→(TheMatrix),
(Carrie)-[:ACTED_IN {roles:['Trinity']}]→(TheMatrix),
```

- The code will be added as illustrated below



```
(JessicaThompson)-[:REVIEWED {summary:'Silly, but fun', rating:65}]→(TheReplacements),
(JamesThompson)-[:REVIEWED {summary:'The coolest football movie ever', rating:100}]→(TheReplacements),
(AngelaScope)-[:REVIEWED {summary:'Pretty funny at times', rating:62}]→(TheReplacements),
(JessicaThompson)-[:REVIEWED {summary:'Dark, but compelling', rating:85}]→(Unforgiven),
(JessicaThompson)-[:REVIEWED {summary:"Slapstick redeemed only by the Robin Williams and Gene Hackman's stellar
performances", rating:45}]→(TheBirdcage),
(JessicaThompson)-[:REVIEWED {summary:'A solid romp', rating:68}]→(TheDaVinciCode),
(JamesThompson)-[:REVIEWED {summary:'Fun, but a little far fetched', rating:65}]→(TheDaVinciCode),
(JessicaThompson)-[:REVIEWED {summary:'You had me at Jerry', rating:92}]→(JerryMaguire)

WITH TomH as a
MATCH (a)-[:ACTED_IN]→(m)←[:DIRECTED]-(d) RETURN a,m,d LIMIT 10;
```

```
$ :play movies
```

The Movie Graph

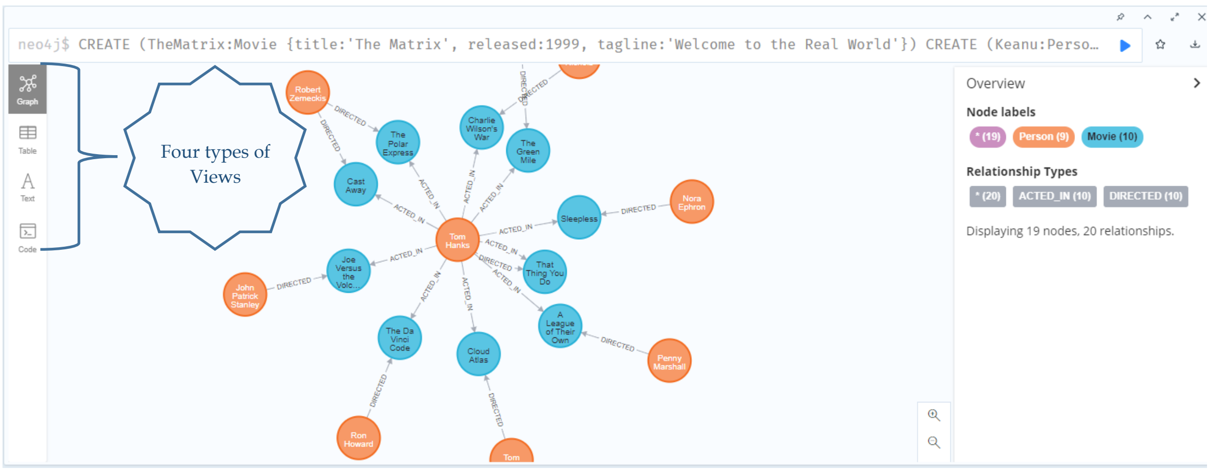
Create

To the right is a giant code block containing a single Cypher query statement composed of multiple CREATE clauses. This will create the movie graph.

1. Click on the code block
2. Notice it gets copied to the editor above ↑
3. Click the editor's play button to execute
4. Wait for the query to finish
   WARNING: This adds data to the current database, each time it is run!

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to the Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Carrie:Person {name:'Carrie-Anne Moss', born:1967})
CREATE (Laurence:Person {name:'Laurence Fishburne', born:1961})
CREATE (Hugo:Person {name:'Hugo Weaving', born:1960})
CREATE (LillyW:Person {name:'Lilly Wachowski', born:1967})
CREATE (LanaW:Person {name:'Lana Wachowski', born:1965})
CREATE (JoelS:Person {name:'Joel Silver', born:1952})
CREATE
(Keanu)-[:ACTED_IN {roles:['Neo']}]→(TheMatrix),
(Carrie)-[:ACTED_IN {roles:['Trinity']}]→(TheMatrix),
```

- To run the code, press **Ctrl + Enter** keys or click on the **Run** icon, which has a shape of a right arrow icon. On the top-right corner of the command line, you will see a visualization of the graph database, as illustrated below.
- .



```
neo4j$ CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to the Real World'}) CREATE (Keanu:Perso…
```

Overview

Node labels

* (19) Person (9) Movie (10)

Relationship Types

* (20) ACTED_IN (10) DIRECTED (10)

Displaying 19 nodes, 20 relationships.

- Click on **Next (to page3)**
- Click on each of the four cyphers and run them:
  - Select the actor named "Tom Hanks"
    MATCH (tom {name: "Tom Hanks"}) RETURN tom
  - Select the movie with the title "cloudAtlas"
    MATCH (cloudAtlas {title: "Cloud Atlas"}) RETURN cloudAtlas

- Select 10 people from the move graph
  MATCH (people:Person) RETURN people.name LIMIT 10

- Select movies released during the 1990s
  MATCH (nineties:Movie) WHERE nineties.released >= 1990 AND nineties.released < 2000 RETURN nineties.title

Exercise
Modify the last cypher to show not only the titles but also the released years.

- Click on **Next (to page4)**
  Finding patterns within the graph
- Click and run the four Cyphers to find patterns
  - List all movies by the actor "Tom Hanks"
    MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(tomHanksMovies) RETURN tom,tomHanksMovies

  - Find out who directed "cloudAtlas"
    MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(tomHanksMovies) RETURN tom,tomHanksMovies

  - Find out who co-acted with "Tom Hanks"
    MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors) RETURN coActors.name

  - Find out how people are related to "cloudAtlas"
    MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"}) RETURN people.name, Type(relatedTo), relatedTo

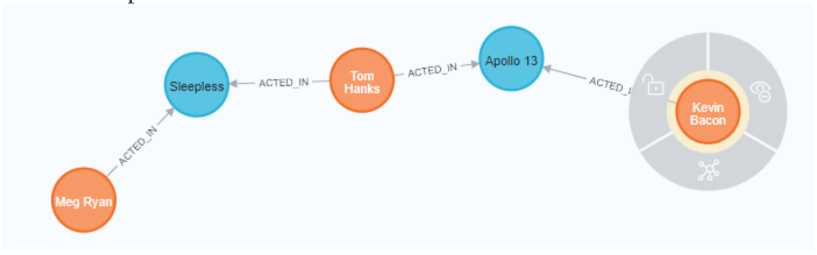
Exercise
Modify the last cypher to show the visualization of the people who are related to the movie 'Cloud Atlas'.

- Click on **Next (to page5)**
  Solve or find out movies and actors a given number of hops away, also find the shortest path between any node (actor) relationships to the node Meg Ryan
- Click and run the four Cyphers to find:
  - actors of up to 4 hops away from "Kevin Bacon"
    MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood) RETURN DISTINCT hollywood

    (*This will return a minimum length of 1 and a maximum length of 4, thus 2 nodes and 1 relationship, 3 nodes and 2 relationships, 4 nodes and 3 relationships, 5 nodes and 4 relationships*)

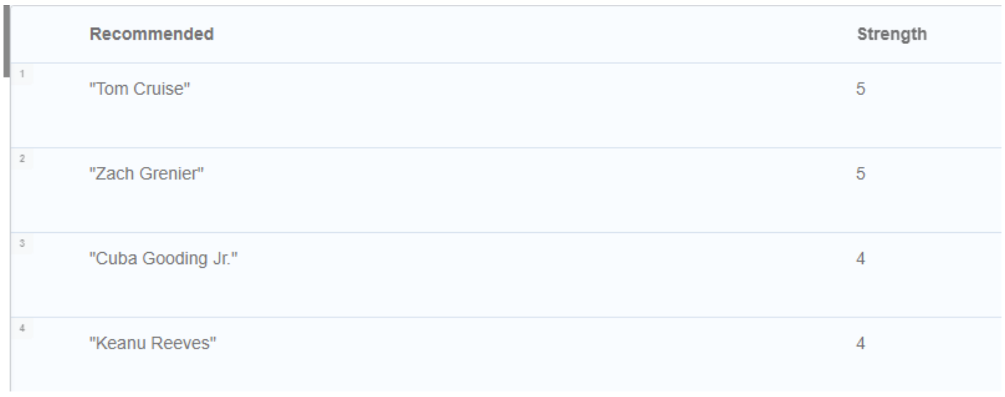
- the shortest path between "Kevin Bacon" and "Meg Ryan"

  ```
  MATCH p=shortestPath(
  (bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"})
  )
  RETURN p
  ```

- Click on **Next (to page6)**
  Recommend – recommend a new co-actor for "Tom Hanks", so to find a potential co-actor it is possible to find immediate neighbors from previous connections.
- Click and run the two Cyphers to find:
  - Actors that "Tom Hanks" has not co-acted before but his co-actors have.

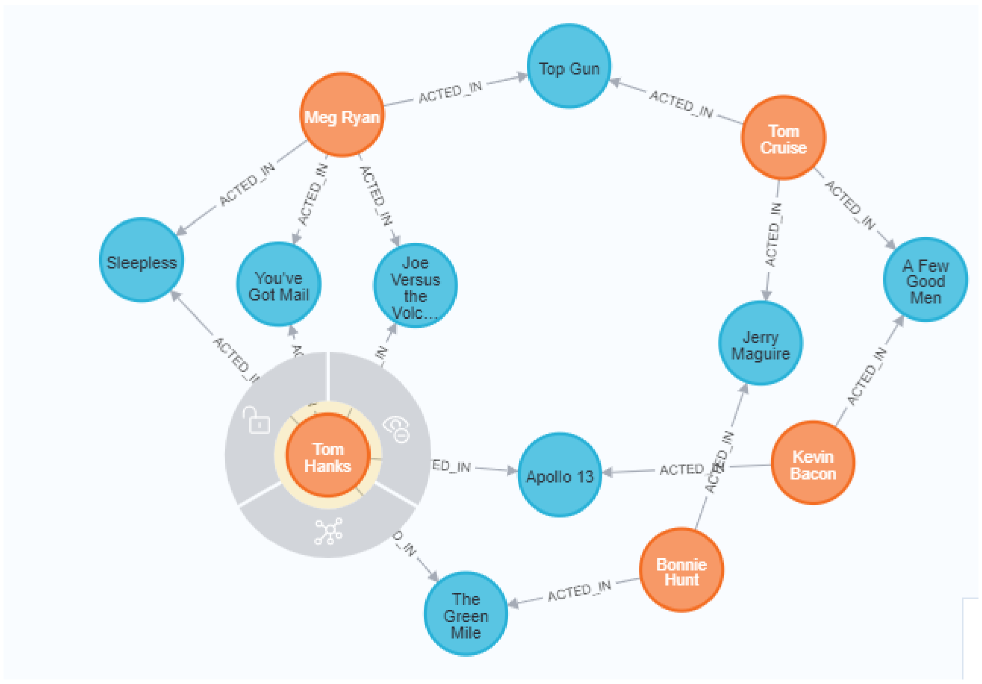
    ```
    MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
    WHERE NOT (tom)-[:ACTED_IN]->()<-[:ACTED_IN]-(cocoActors) AND tom <> cocoActors
    RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
    ```

	Recommended	Strength
1	"Tom Cruise"	5
2	"Zach Grenier"	5
3	"Cuba Gooding Jr."	4
4	"Keanu Reeves"	4

  - Find out who can introduce "Tom Hanks" to potential co-actor, i.e., "Tom Cruise"

    ```
    MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
    RETURN tom, m, coActors, m2, cruise
    ```



- Here you can see that Meg Ryan, who has acted in three moves with Tom Hanks, Bonnie Hunt, and Kevin Bacon, could introduce Tom Hanks to Tom Cruise.

- Click on **Next (to page7) – Leave it and continue to the next section.**

! DO NOT DELETE THE NODES AND THEIR RELATIONSHIPS, SO DO NOT RUN THE CYPHER MATCH (n) DETACH DELETE n

YOU WILL USE THE DATA IN THE NEXT SECTION!

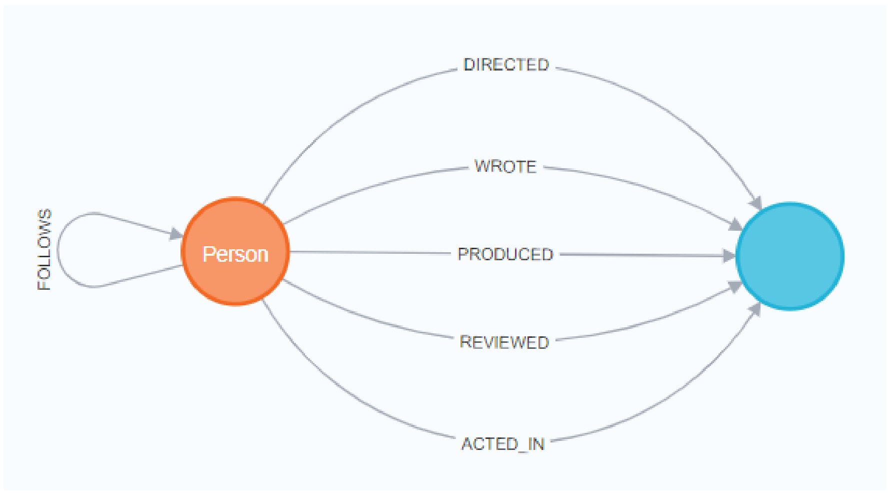
In the next section, you will learn how to write parts of cypher by yourself by example.

1.2. Writing your cypher

In this section, you will learn the basics for writing advanced cypher. To visualize the schema of the graph database use the cypher:

```
CALL db.schema.visualization
```

The graph schema and its relationships will be created.



1.2.1. Query performance and memory limitations

Some of the cypher codes might return or result in several lines of rows, and even cypher codes might return an infinite list of rows. In these cases, you need to (1) use the **LIMIT** keyword to determine how many rows should be returned, (2) create indexes for improved search performance, and finally (3) extend the heap memory size of your computer.

1. **LIMIT** is followed by a positive integer or any expression that ends up giving a positive integer (but the expression should not use nodes and relationships).

   Example – LIMIT

   ```
   MATCH (n:Movie)
   RETURN n.title,n.released,n.tagline
   ORDER BY n.title
   LIMIT 5
   ```

2. **Indexes** – create two indexes on the movie title and the person name.

   ```
   CREATE  INDEX index_movie_title IF NOT EXISTS
   FOR (m:Move) ON (m.title);

   CREATE  INDEX index_person_name IF NOT EXISTS
   FOR (p:Person) ON (p.name);
   ```

   To check if the indexes are already created or not, use the following cypher:

   ```
   CALL db.indexes
   ```

   DO NOT DELETE THE INDEXES, but the syntax to delete indexes is:

   ```
   DROP INDEX [index_NAME]
   ```

3. **Increasing heap memory size**
   - Form your Neo4j Desktop, open the **Settings** for the database, using the drop-down arrow next to the **Open** button.
   - Edit the property described below to 2GB as illustrated:
     dbms.memory.heap.max_size=2G.
   - Click **Apply,** which will restart the DBMS.

1.2.2. CASE-expressions

CASE-expressions – is used to compare an expression against a number of values (keywords used are CASE, WHEN, END, THEN).

```
Syntax:
CASE expr1
        WHEN value THEN result1
        WHEN value2 THEN result2
        …
        [ELSE default]
END
```

Example – CASE expressions
Write a cypher that will return "Best Actor" if the actor is Meg Ryan, Tom Cruise or Tom Hanks, and "No Award" otherwise.

Solution:

```
MATCH (n)
RETURN Distinct n.name,
   CASE n.name
      WHEN 'Tom Cruise' THEN 'Best Actor'
      WHEN 'Meg Ryan' THEN 'Best Actor'
      WHEN 'Tom Hanks' THEN 'Best Actor'
      ELSE 'No Award'
   END AS result
```

If you want to show nodes with names while discarding nodes without a name, use the following cypher.

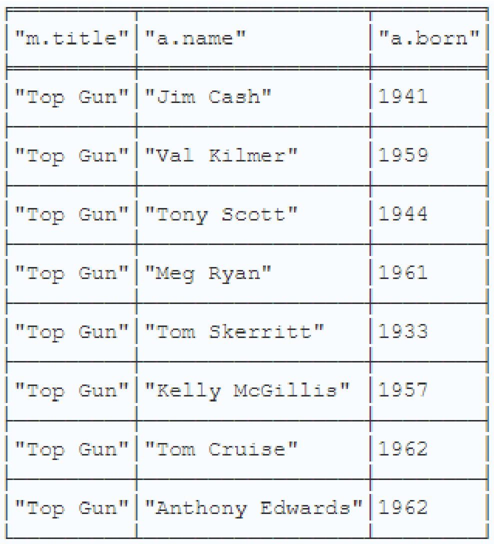
```
MATCH (n)
WHERE n.name IS NOT NULL
RETURN Distinct n.name,
   CASE n.name
      WHEN 'Tom Cruise' THEN 'Best Actor'
      WHEN 'Meg Ryan' THEN 'Best Actor'
      WHEN 'Tom Hanks' THEN 'Best Actor'
      ELSE 'No Award'
   END AS result
```

1.2.3. OPTIONAL MATCH -Clause

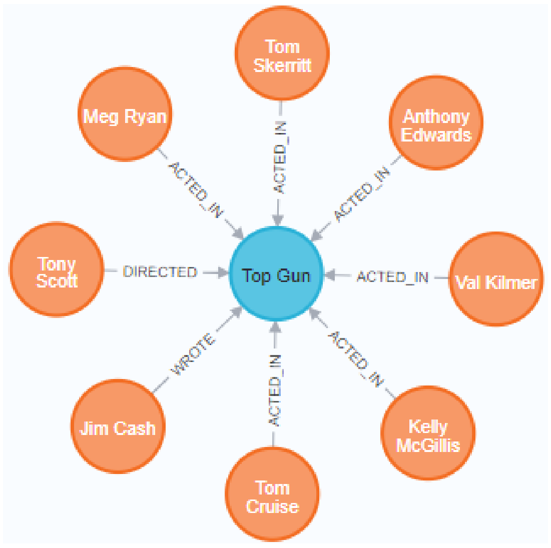
The **OPTIONAL MATCH** -Clause is similar to the **OUTER JOIN** of **SQL**, and it is used to search for patterns, and if the pattern is unavailable, it will return **NULL**. For example, if you want to look for the Movie with the title 'Top Gun', and if you want to optionally search for all Persons associated with the Movie optionally, then you should use the **OPTIONAL MATCH**-clause.

Example – OPTIONAL MATCH-Clause

```
MATCH (m:Movie {title: 'Top Gun'})
OPTIONAL MATCH (m)<-[r]-(a)
RETURN m.title,a.name,a.born
```



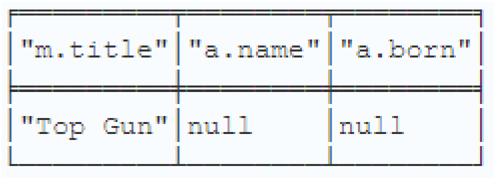
"m.title"	"a.name"	"a.born"
"Top Gun"	"Jim Cash"	1941
"Top Gun"	"Val Kilmer"	1959
"Top Gun"	"Tony Scott"	1944
"Top Gun"	"Meg Ryan"	1961
"Top Gun"	"Tom Skerritt"	1933
"Top Gun"	"Kelly McGillis"	1957
"Top Gun"	"Tom Cruise"	1962
"Top Gun"	"Anthony Edwards"	1962



If the **OPTIONAL MATCH** does not return anything, then you will have null instead. For example, in the previous example, the relationship pattern is from actor to movie, but if you reverse the relationship from movie to the actor, then it will return nothing.

Example – OPTIONAL MATCH-Clause

```
MATCH (m:Movie {title: 'Top Gun'})
OPTIONAL MATCH (m)-[r]->(a)
RETURN m.title,a.name,a.born
```



"m.title"	"a.name"	"a.born"
"Top Gun"	null	null

1.2.4. WITH – Clause

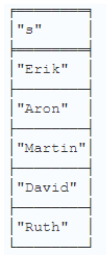
The WITH-Clause is used to chain query parts together. In other words, it is used for piping the result of a preceding point with succeeding parts. In this case, we can use the name (or names) of the variable(s) to be piped and "*" to include all variables within the scope to the next lines of your cypher.

Example – WITH-Clause

```
MATCH (actor)-[]-(coActor)
WHERE actor.name IS NOT NULL AND coActor.name IS NOT NULL
WITH actor,coActor, toUpper(coActor.name) AS CoActor
WHERE CoActor STARTS WITH 'J' OR CoActor STARTS WITH 'A'
RETURN actor.name, coActor.name
```

1.2.5. UNWIND – Clause

The WITH-Clause is used to convert a liSt into a sequential rows.



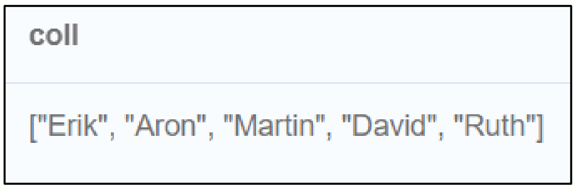
"s"
"Erik"
"Aron"
"Martin"
"David"
"Ruth"

Example – UNWIND-Clause

```
WITH ['Erik','Aron','Martin','David','Ruth'] AS students
UNWIND students AS pupil
RETURN pupil
```

Example – UNWIND-Clause (*collect ()* is a function that does the opposite of UNWIND-clause, i.e., convert rows to list)

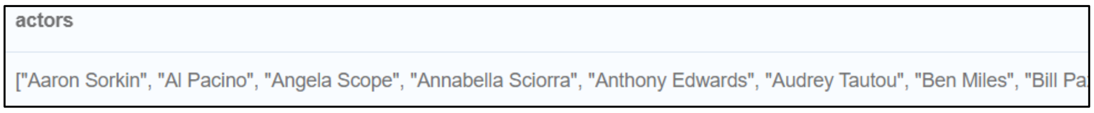
```
WITH ['Erik','Aron','Martin','David','Ruth'] AS students
UNWIND students AS pupil
WITH collect(pupil) AS coll
RETURN coll
Or
WITH ['Erik','Aron','Martin','David','Ruth'] AS students
UNWIND students AS s
RETURN collect(s) AS coll
```



coll
["Erik", "Aron", "Martin", "David", "Ruth"]

Example – UNWIND-Clause (*collect ()* is a function that does the opposite of UNWIND-clause, i.e., convert rows to list)

```
MATCH (n:Person)
With collect(n.name) AS actors
RETURN actors
```



actors
["Aaron Sorkin", "Al Pacino", "Angela Scope", "Annabella Sciorra", "Anthony Edwards", "Audrey Tautou", "Ben Miles", "Bill Pa

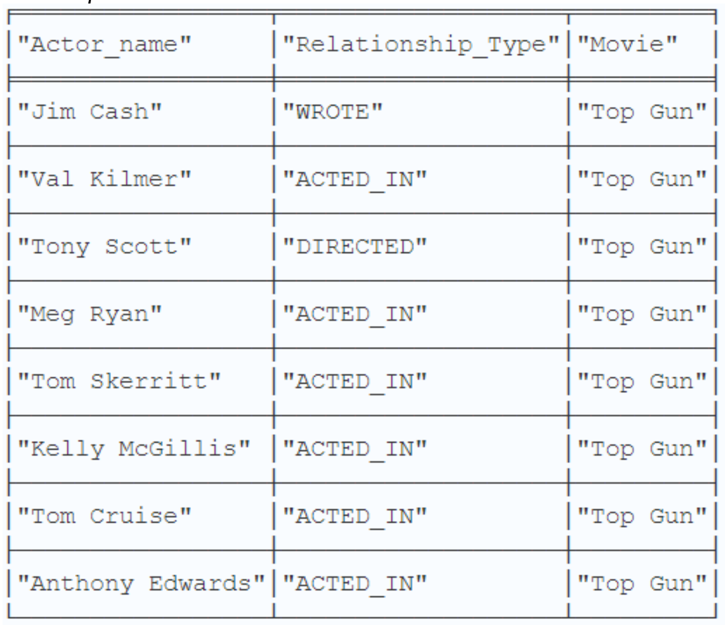
1.2.6. WHERE and ORDER BY

WHERE is not a clause; instead, it is part of other clauses, especially MATCH, OPTIONAL MATCH. On the contrary, ORDER BY is a sub-clause of RETURN or WITH for specifying sorting parameters for the returned rows.

Example – WHERE

- Operations used (NOT, AND, OR, XOR, <, >, <=, >=, <>, etc)
- Operations used in string operations (CONTAINS, ENDS WITH, and STARTS WITH followed by a string literal)
- Negation use NOT keyword
- Filtering on patterns: Example
    - WHERE (x)-[*]->(y) ← *here it will eliminate paths that do not include paths connecting x and y.*
    - WHERE actor.name IN ['Erik','Aron','Martin','David','Ruth'] ← *here it will include actor's whose names are included in the list*
    - WHERE (actor)<- - (topgun) ← *look for actors who have an incoming relationship from the movie Top Gun*
    - WHERE NOT (actor)<- - (topgun) ← *look for actors who do not have an incoming relationship from the movie Top Gun*
    - Select actors names, relationship types they have with the movie "top gun", that begins with W or D or A

      MATCH (actor: Person)- [rel]-> (topgun:Movie{title:'Top Gun'})
      WHERE type(rel) =~ 'W.*' OR type(rel) =~ 'D.*' OR type(rel) =~ 'A.*'
      RETURN actor.name AS Actor_name, type(rel) AS Relationship_Type, topgun.title AS Movie
      ← *it returns actor name, relationship type beginning with either W or D or A for movie title Top Gun*



```
"Actor_name"      |"Relationship_Type"|"Movie"
"Jim Cash"        |"WROTE"            |"Top Gun"
"Val Kilmer"      |"ACTED_IN"         |"Top Gun"
"Tony Scott"      |"DIRECTED"         |"Top Gun"
"Meg Ryan"        |"ACTED_IN"         |"Top Gun"
"Tom Skerritt"    |"ACTED_IN"         |"Top Gun"
"Kelly McGillis"  |"ACTED_IN"         |"Top Gun"
"Tom Cruise"      |"ACTED_IN"         |"Top Gun"
"Anthony Edwards" |"ACTED_IN"         |"Top Gun"
```

- EXISTS – subquery under WHERE
    - MATCH (per:Person)-[:WROTE]->(mov:Movie)
      WHERE EXISTS {
        MATCH (per{name : 'Lilly Wachowski'})-[:WROTE]->(mov:Movie)
      }
      RETURN per.name AS name, mov.title as Movie
      ← *it returns movies with title written by Lilly Wachowski*

Example – ORDER BY

- Ascending
    - ORDER BY actor.name ← *sort by actor name in ascending order*

    - ORDER BY actor.name, actor.born ←*sort by actor name and then by born in ascending order*
- Descending
    - ORDER BY actor.name DESC ←*sort by actor name in descending order*

1.2.7. Creating paths

A path in the Neo4j cypher or graph query language is a variable that holds a directed sequence of relationships or connections between nodes. It is crucial to keep the in mind while path matching as the number of path patterns can be very large. Therefore, it is suggested to have a constraint check. Let us consider the three options of pattern matching, namely (1) homomorphism, (2) node isomorphism, and (3) relationship isomorphism.

Homomorphism

- Homomorphism enforces no constraint on either the nodes or relationships.

Example: Find all relationships between any nodes could between any of the node categories, using named path or path variable p.

```
MATCH p = ()-[]->()
RETURN p, nodes(p)
```

Or

```
MATCH p = ()-[rel]->()
RETURN p, type(rel), nodes(p)
```

If you want to restrict the number of results returned, for example, to get only three results use the following cypher

```
MATCH p = ()-[*3]->()
RETURN p, nodes(p)
```

Or

```
MATCH p = ()-[rel]->()
RETURN p, type(rel) , nodes(p)
Limit 3
```

Note: generally, if you get to review all the returned results, the result will contain duplicate relationships between nodes as there is no restriction. For example, if two nodes a and b have a relationship type, the result will return [a, b] and [b, a].

Node isomorphism

- Node isomorphism enforces a constraint to ensure that no node is returned twice for a specific path matching record.

Example: Find all relationships between actor 'Lilly Wachowski' and movies she wrote

```
MATCH (per{name : 'Lilly Wachowski'})-[:WROTE]->(mov:Movie)
RETURN per.name AS name, mov.title as Movie
```

Relationship isomorphism

- Relationship isomorphism enforces a constraint to ensure that a relationship is only returned not more than once for a given path matching record. Relationship isomorphism is used to do path matching focusing on the relationship. For example, if you want to find out networks of relationships, relationship rings.

Example: Find a person(s) who follow another group of person(s) following at least a person(s).

```
MATCH (p1:Person)-[:FOLLOWS]->(p2:Person)-[:FOLLOWS]->(p3:Person)
```

```
RETURN p1.name as P1, p2.name AS P2_followed_by_P1, p2.name AS followed_by_P2_WHO_IS_f
ollower_of_2
```

```
╒═════════════╤═══════════════════╤═════════════════════════════════════╕
│"P1"         │"P2_followed_by_P1"│"followed_by_P2_WHO_IS_follower_of_2"│
╞═════════════╪═══════════════════╪═════════════════════════════════════╡
│"Paul Blythe"│"Angela Scope"     │"Angela Scope"                       │
└─────────────┴───────────────────┴─────────────────────────────────────┘
```

Example: Find a node(s) who follow or have any type of relationship with another group of node(s) following or having any type of relationship with at least a node(s).

```
MATCH (p1)-[]->(p2)-[]->(p3)
RETURN DISTINCT p1.name as P1, p2.name AS P2_followed_by_P1, p2.name AS followed_by_P2_
WHO_IS_follower_of_2
```

```
╒════════════════╤═══════════════════╤═════════════════════════════════════╕
│"P1"            │"P2_followed_by_P1"│"followed_by_P2_WHO_IS_follower_of_2"│
╞════════════════╪═══════════════════╪═════════════════════════════════════╡
│"Paul Blythe"   │"Angela Scope"     │"Angela Scope"                       │
├────────────────┼───────────────────┼─────────────────────────────────────┤
│"James Thompson"│"Jessica Thompson" │"Jessica Thompson"                   │
├────────────────┼───────────────────┼─────────────────────────────────────┤
│"Angela Scope"  │"Jessica Thompson" │"Jessica Thompson"                   │
└────────────────┴───────────────────┴─────────────────────────────────────┘
```

Creating paths other examples

Example: query all paths having three nodes and two relations, where the middle node receives an incoming relationship from both nodes.

```
MATCH p = (a)-[]->(m)<-[]-(d)
RETURN p, a, m, d
```

Example: query all paths having three nodes and two relations, where without restricting the direction of existing relationship.

```
MATCH p = (a)-[]-(m)-[]-(d)
RETURN p, a, m, d
```

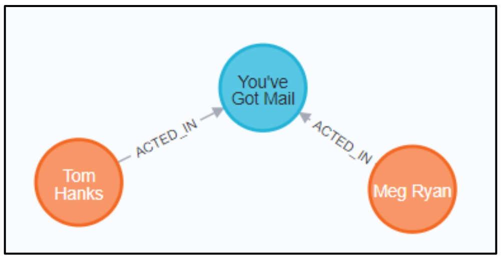
Example: if you want to count the number of paths having the relationship pattern ()-[]-()-[]-(), use the following cypher.

```
MATCH p= (a)-[]-(m)-[]-(d)
RETURN p, a, m, d, count(*)
```

Shortest paths between two known nodes.

Example: Find the shortest path between Tom hanks and Meg Ryan, and restrict to search for a max of 10 relations between them.

```
MATCH
  (tom:Person {name: 'Tom Hanks'}),
  (meg:Person {name: 'Meg Ryan'}),
  p = shortestPath((tom)-[*..10]-(meg))
RETURN p
```



Shortest paths between a known node and all other nodes.

Example: Find the shortest path between Tom Hanks and any other node in the graph database, limit the result so that you can get fewer results.

```
MATCH
  (tom:Person {name: 'Tom Hanks'}),
  (x:Person),
  p = shortestPath((tom)-[*..15]-(x))
  WHERE x<>tom
RETURN p
LIMIT 3
```

1.2.8. Aggregate functions

The following are aggregation functions:

- count (n)
- max (n)
- min (n)
- collect (n)

Example – count()

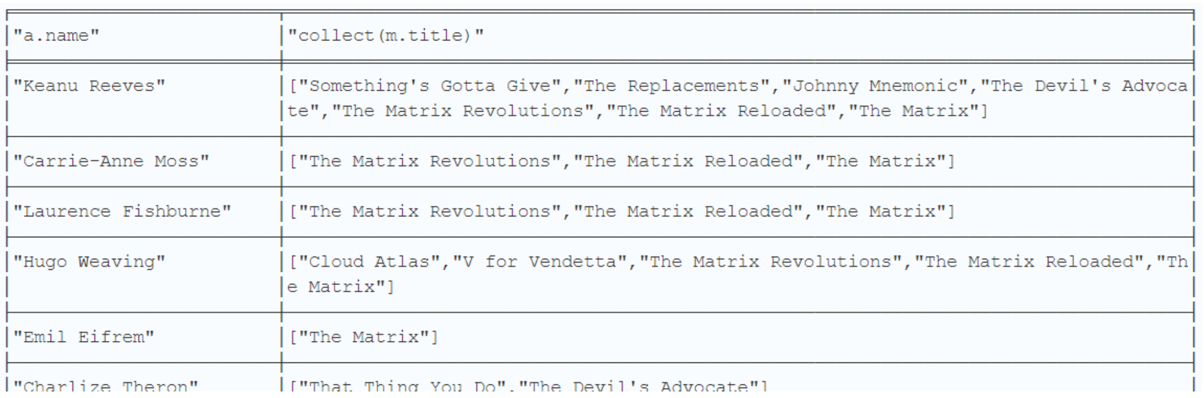
Find all actors and the number of movies they acted in.

```
MATCH (a)-[:ACTED_IN]->(m)
RETURN a.name, count(m)
Or
MATCH (a)-[:ACTED_IN]->(m)
RETURN a.name, count(*)
```

Example – collect ()

Find all actors and the list of movies they acted in.

```
MATCH (a)-[:ACTED_IN]->(m)
RETURN a.name, collect(m.title)
```



```
│"a.name"              │"collect(m.title)"                                                        │
╞══════════════════════╪══════════════════════════════════════════════════════════════════════════╡
│"Keanu Reeves"        │["Something's Gotta Give","The Replacements","Johnny Mnemonic","The Devil's Advoca│
│                      │te","The Matrix Revolutions","The Matrix Reloaded","The Matrix"]          │
├──────────────────────┼──────────────────────────────────────────────────────────────────────────┤
│"Carrie-Anne Moss"    │["The Matrix Revolutions","The Matrix Reloaded","The Matrix"]             │
├──────────────────────┼──────────────────────────────────────────────────────────────────────────┤
│"Laurence Fishburne"  │["The Matrix Revolutions","The Matrix Reloaded","The Matrix"]             │
├──────────────────────┼──────────────────────────────────────────────────────────────────────────┤
│"Hugo Weaving"        │["Cloud Atlas","V for Vendetta","The Matrix Revolutions","The Matrix Reloaded","Th│
│                      │e Matrix"]                                                                │
├──────────────────────┼──────────────────────────────────────────────────────────────────────────┤
│"Emil Eifrem"         │["The Matrix"]                                                            │
├──────────────────────┼──────────────────────────────────────────────────────────────────────────┤
│"Charlize Theron"     │["That Thing You Do","The Devil's Advocate"]                              │
```

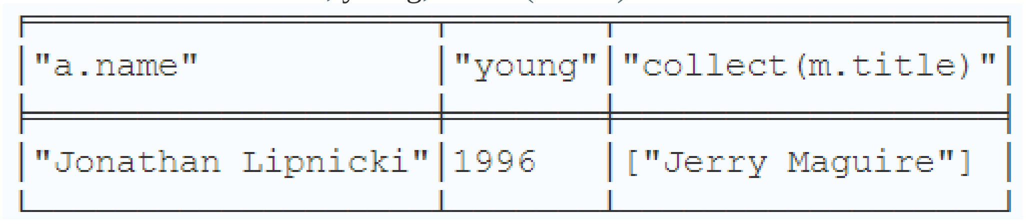
Example – max ()

Find the youngest actor and the list of movies she/he acted in.

```
MATCH (n:Person)
WITH max(n.born) AS young
OPTIONAL MATCH (a)-[:ACTED_IN]->(m)
WHERE a.born = young
RETURN a.name, young, collect(m.title)
```

Or

```
MATCH (n:Person)
WITH max(n.born) AS young
MATCH (a)-[:ACTED_IN]->(m)
WHERE a.born = young
RETURN a.name, young, collect(m.title)
```



```
│"a.name"            │"young"│"collect(m.title)"│
╞════════════════════╪═══════╪══════════════════╡
│"Jonathan Lipnicki" │1996   │["Jerry Maguire"] │
└────────────────────┴───────┴──────────────────┘
```

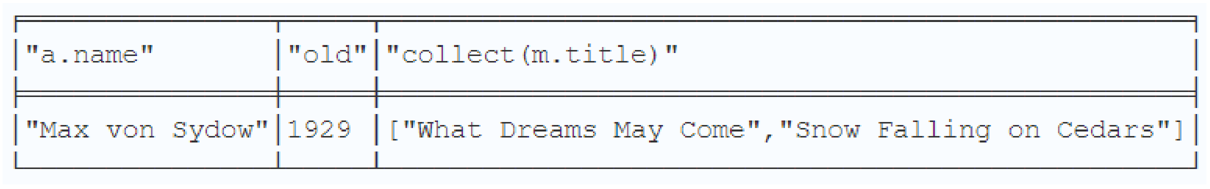
Example – min ()

Find the oldest actor and the list of movies she/he acted in.

```
MATCH (n:Person)
WITH min(n.born) AS young
OPTIONAL MATCH (a)-[:ACTED_IN]->(m)
WHERE a.born = young
RETURN a.name, young, collect(m.title)
```

Or

```
MATCH (n:Person)
WITH min(n.born) AS young
MATCH (a)-[:ACTED_IN]->(m)
WHERE a.born = young
RETURN a.name, young, collect(m.title)
```



```
╒════════════════╤═════╤═══════════════════════════════════════════════╕
│"a.name"        │"old"│"collect(m.title)"                             │
╞════════════════╪═════╪═══════════════════════════════════════════════╡
│"Max von Sydow" │1929 │["What Dreams May Come","Snow Falling on Cedars"]│
└────────────────┴─────┴───────────────────────────────────────────────┘
```

Finally, a visualization tool called Bloom (see Appendix 1) is far more advanced than the Neo4j Browser. However, it only works directly on our VM with applying configurations. Thus, you can try it on your machines if you have time.

2. References

Sasaki, B. M., Chao, J., & Howard, R. (2018). Graph databases for beginners. *Neo4j*.

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".

https://neo4j.com/docs/bloom-user-guide/current/

https://neo4j.com/docs/cypher-manual/current/

3. Appendix 1 - Visualization using Bloom

3.1.1. Add a calculated property (PageRank) on all Person nodes

This section presents how Page Rank is calculated and added to each node as a property for every person node. Thus you will write it to Person's graph as an additional property. PageRank is a centrality measuring algorithm used to measure the importance of nodes in a graph, and the higher the PageRank value of a node in a network, the most important the node is. Page rank and subgraphs are not discussed in detail in this tutorial, yet, you will learn them in detail in the next tutorial.

Step 1: Create a subgraph

Here we present examples of subgraph creation cypher, where we project the movie graph to a subgraph called subgraphPerson with:

1. The node label 'Person' and any type of relationship in Cypher_A.
2. The node label 'Person' with the relationship type 'ACTED_IN' in Cypher_B.
3. The node label 'Person' with the relationship type 'DIRECTED_IN' in Cypher_C.

USE THE FIRST CYPHER, Cypher_A, for now.

Cypher_A

```
call gds.graph.create.cypher('subgraphPerson',
    'Match (p:Person) RETURN ID(p) AS id',
    'Match (p:Person)-->()<--(p2:Person) Return ID(p) AS source, ID(p2) AS
    target')
```

Cypher_B

```
call gds.graph.create.cypher('subgraphPerson',
    'Match (p:Person) RETURN ID(p) AS id',
    'Match (p:Person)-[:ACTED_IN]->()<-[:ACTED_IN]-(p2:Person) Return ID(p)
    AS source, ID(p2) AS target')
```

Cypher_B

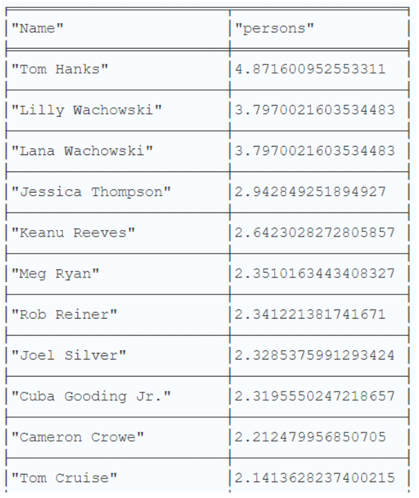
```
call gds.graph.create.cypher('subgraphPerson',
    'Match (p:Person) RETURN ID(p) AS id',
    'Match (p:Person)-[:DIRECTED_IN]->()<-[: DIRECTED _IN]-(p2:Person)
    Return ID(p) AS source, ID(p2) AS target')
```

Step 2: Write the PageRank value to the subgraph

```
call gds.pageRank.write('subgraphPerson',{writeProperty:'pageRank'})
```

Query your graph so that you see the pageRank property is actually written to the graph

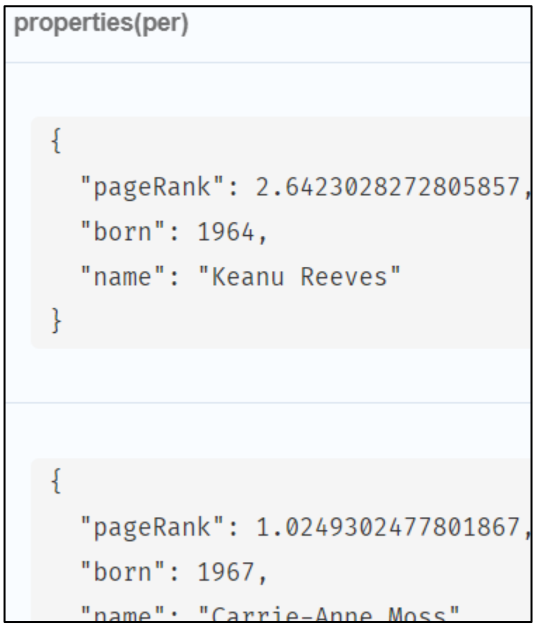
```
MATCH (per:Person) RETURN per.name as Name, per.pageRank as persons order by persons  desc
```



"Name"	"persons"
"Tom Hanks"	4.871600952553311
"Lilly Wachowski"	3.7970021603534483
"Lana Wachowski"	3.7970021603534483
"Jessica Thompson"	2.942849251894927
"Keanu Reeves"	2.6423028272805857
"Meg Ryan"	2.3510163443408327
"Rob Reiner"	2.341221381741671
"Joel Silver"	2.3285375991293424
"Cuba Gooding Jr."	2.3195550247218657
"Cameron Crowe"	2.212479956850705
"Tom Cruise"	2.1413628237400215

Check if the property created is added

```
MATCH (per:Person) RETURN properties(per)
```



properties(per)

```
{
  "pageRank": 2.6423028272805857,
  "born": 1964,
  "name": "Keanu Reeves"
}

{
  "pageRank": 1.0249302477801867,
  "born": 1967,
  "name": "Carrie-Anne Moss"
```

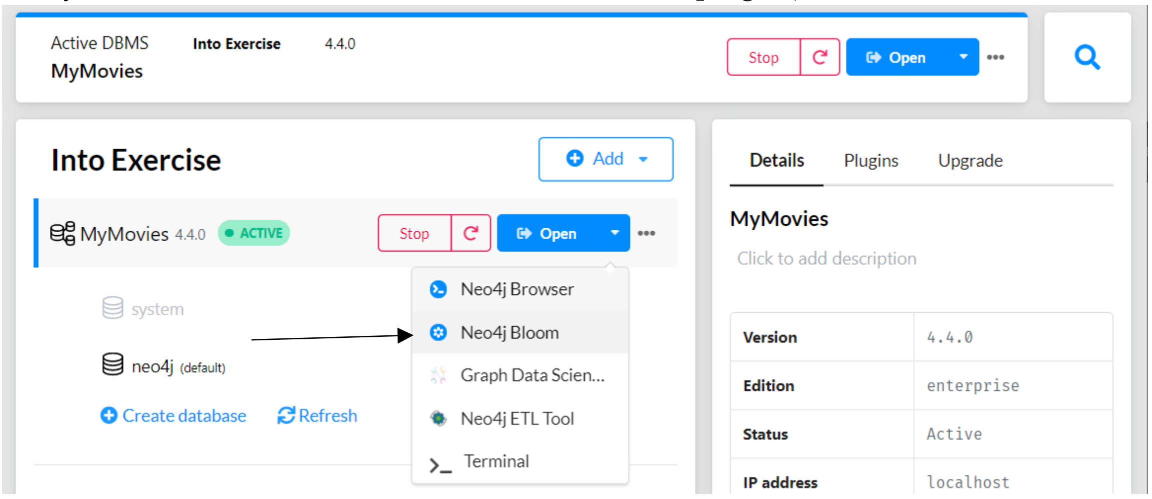
3.1.2. Bloom App

Bloom is an app that is shipped along with the Neo4j Desktop version in the latest versions.

3.1.3. Loading and visualization of data using bloom

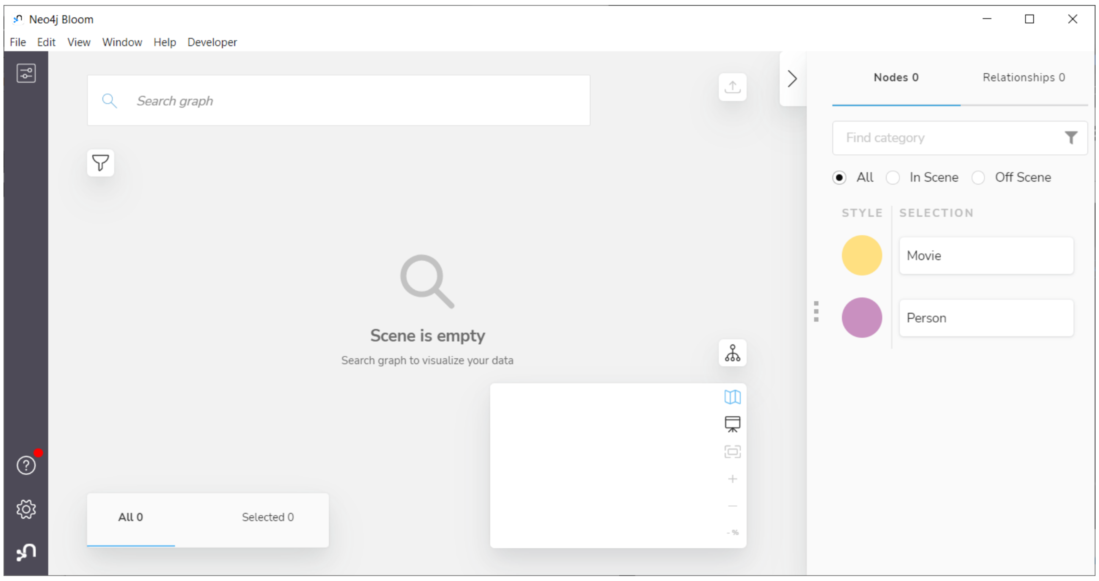
Bloom will use the indexes already created for higher performance visualization and navigation. Bloom will also make statistical analyses on labels and properties of the graph database.

- Go to Neo4j Desktop, click on the down arrow on the Open button and click on Neo4j Bloom (make sure that your database has Graph Data Science Library and if you want to do more advanced tasks, the APOC plugins).



Version	4.4.0
Edition	enterprise
Status	Active
IP address	localhost

- The Neo4j Bloom Windows will be opened as illustrated below.

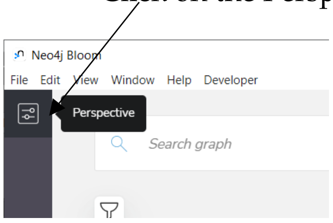


3.1.4. Creating perspectives

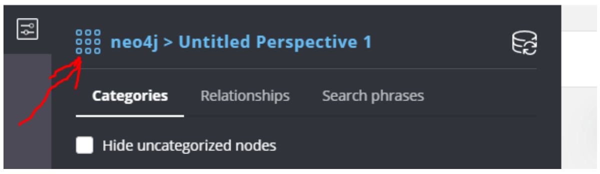
In Neo4j Bloom, Perspectives are used to define domains or business views from graph databases. Thus, providing the functionality to create business views from different contexts of the same graph data.

To create perspectives

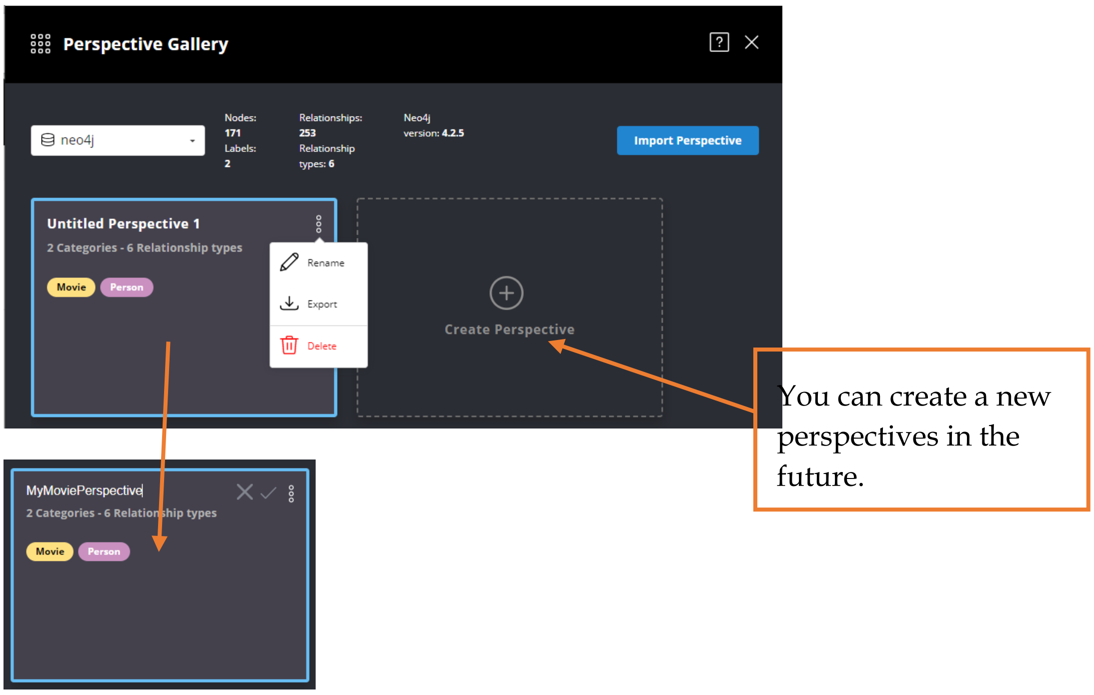
- Click on the Perspective icon on the left corner



- Click on the perspective title (Untitled Perspective 1) as indicated below

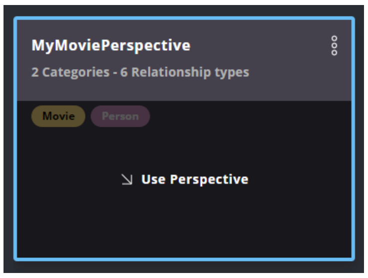


- You will see that you will have two categories (nodes based on their labels) and six relationships. Rename the perspective to **MyMoviePerspective** by clicking on the three vertical dots (ellipsis) and clicking on *Rename.*

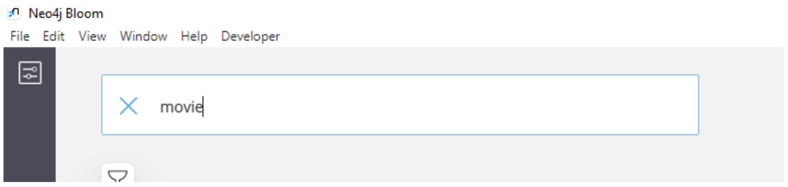


You can create a new perspectives in the future.

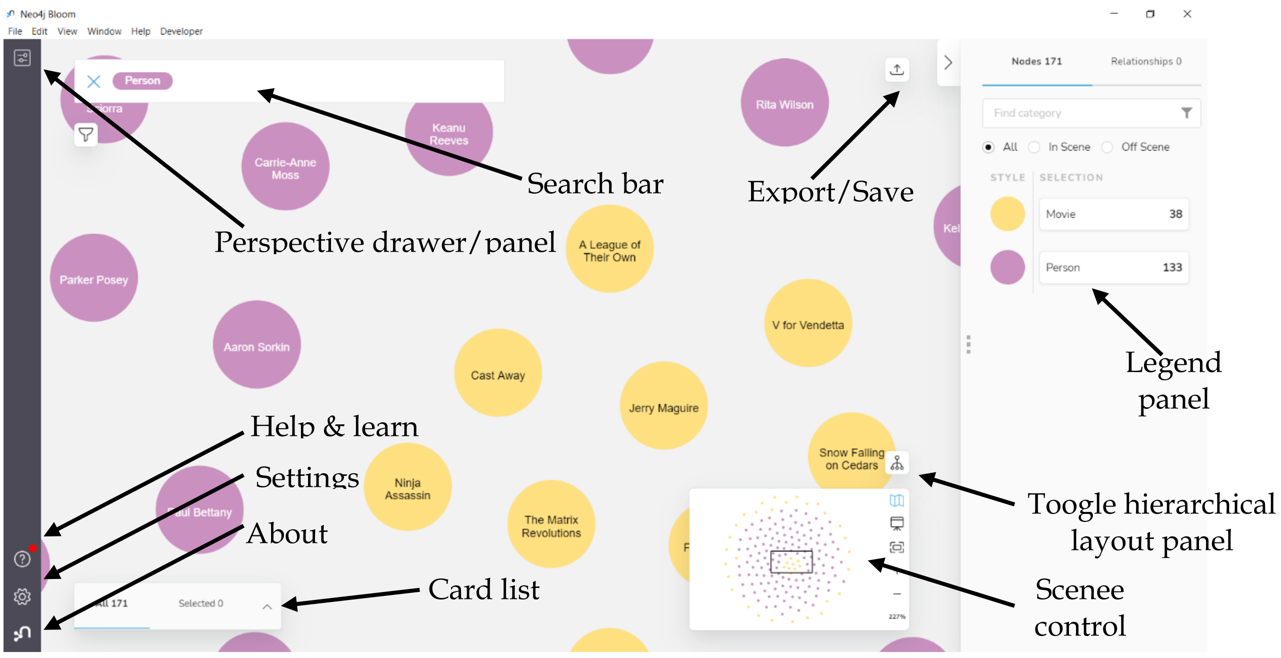
- Hover over the middle of the perspective, *MyMoviePerspective* and click on **Use Perspective**



- Under the search graph box, enter movie -> then press the Enter, here nodes for movies will be generated, and Bloom will give a sensible name to the caption of the nodes using existing properties of the nodes.



- Repeat the search for the person label, the snapshot below illustrates parts of the Neo4j Bloom Window

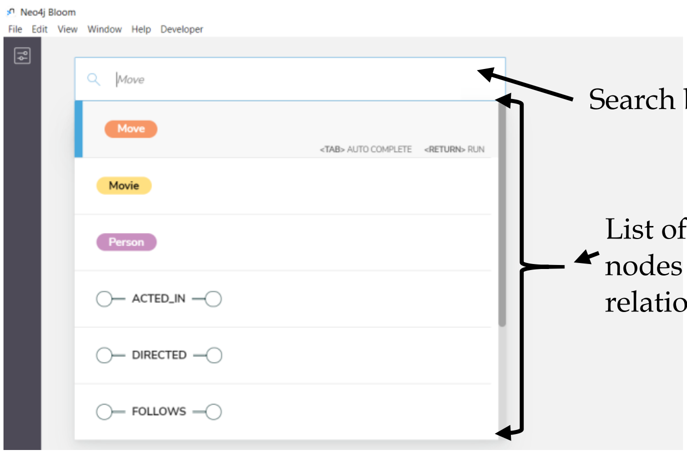


Clearing perspective

- Right click in Bloom free space
- Click on **Clear Scene**

Searching for graph is easy with Bloom

- When you click inside the search graph box you will get a dropdown list of options, see the snapshot below.

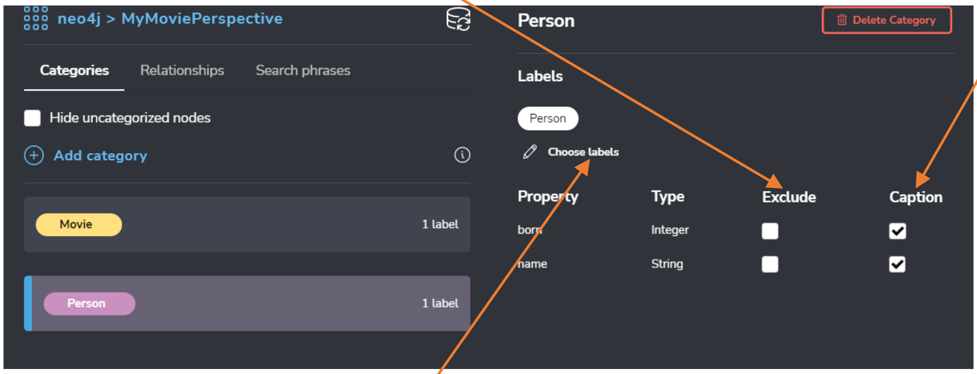

Neo4j Bloom
File Edit View Window Help Developer
Move
Move
<TAB> AUTO COMPLETE <RETURN> RUN
Movie
Person
ACTED_IN
DIRECTED
FOLLOWS
Search bar or searchgraph box
List of options includes nodes and available relationships

- You can select any of this options and press enter to fire visualizations according to the pattern chosen.
- You can also enter Person and wait a bit until you see other relevant search options in the drop-down list.

3.1.5. Customizing Neo4j Bloom visualization

To customize the caption of the node.

- Click on the perspective icon.
- Select Person under **Categories** -> you can now check or uncheck on the **Caption** checkboxes or **Exclude** boxes to hide or show. Check both captions as illustrated below.



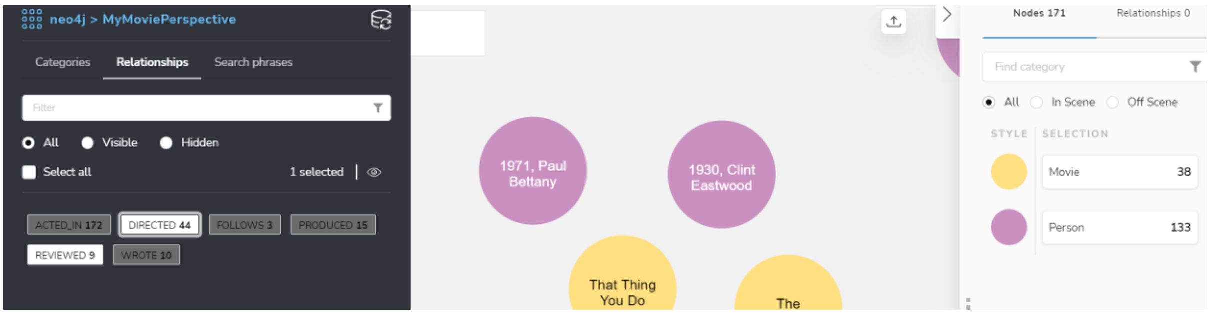
Property	Type	Exclude	Caption
born	Integer	☐	✓
name	String	☐	✓

To add (change) caption labels
(If you have more than one caption for a node).

- Click on **Choose Labels** under **Labels**

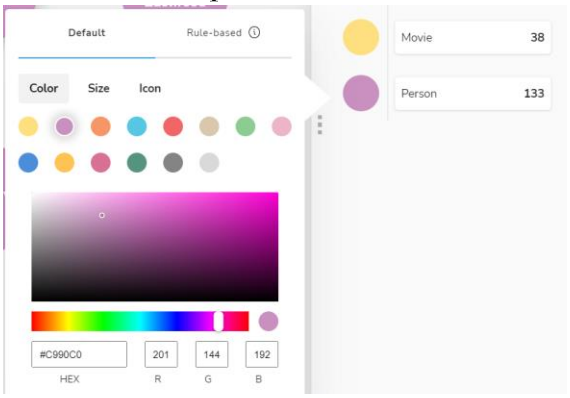
To hide or unhide relationships

- Click on **Relationships, and you** can see available relationships.
- Click on any of the relationships and click on the Icon with an eye symbol. The grey ones are hidden relationships, as illustrated below. For now, leave all relationships unhidden.
- The radio buttons, All, Visible, and Hidden, allow you to filter based on visibility.

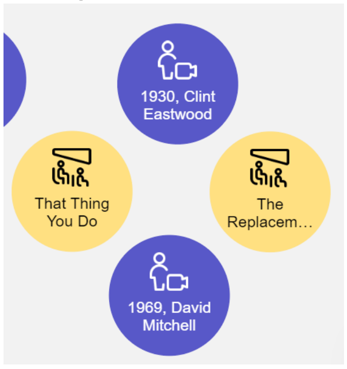


Customizing node using default options.

- On the right panel of the Neo4j Bloom, click on the **Nodes** tab,
- Select *Person*, here you can change Icon visualization using **Color**, **Size**, and **Icon** under **Default**
- Select **Color** – set Red to 88, Green to 88, and Blue to 200
- Select **Size** – leave the default value as it is
- Select an **Icon** – under the **Search for icon** box, enter "person", and you will get a number of icons that are related to the person keyword, select any Icon that you feel is relevant or expressive.



- Change the color, size and icon for Movie node:



Customizing node using Rule-based options

Rule-based visualization, for example change the color of all Person nodes based on a certain rule.

Example:

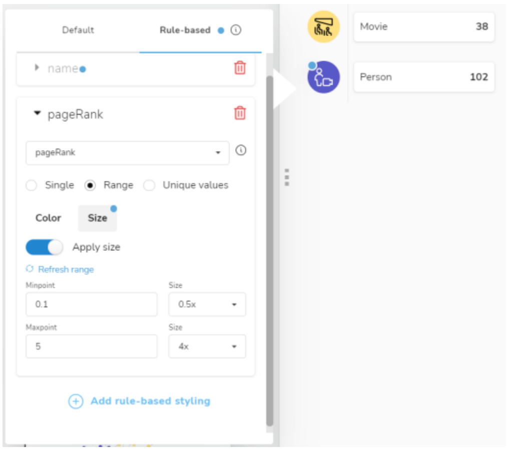
Change the color of person nodes to blue if their name begins with j.

- Click on **Rule-based** tab in Neo4j Perspective
- Under **name** select *name* property
- Under Select property key → select **starts with** text box
- Enter J under **Set property key**
- Choose the blue color
- Check on **Apply size** option

Example:

Change the size of person nodes based on their pageRank value.

- Click on **Rule-based** tab in Neo4j Perspective
- Click on **(+) Add rule-based styling**
- Under **name** select *pageRank* property
- Select **Range** option
- Enter 0.1 under **Minpoint**, and **Size** enter 0.5x
- Enter 5 under **Maxpoint**, and **Size** enter 4x
- Check on **Apply size** option


Default | Rule-based

name

pageRank

pageRank

( ) Single [✓ Range] ( ) Unique values

Color | Size

Apply size: on

Refresh range

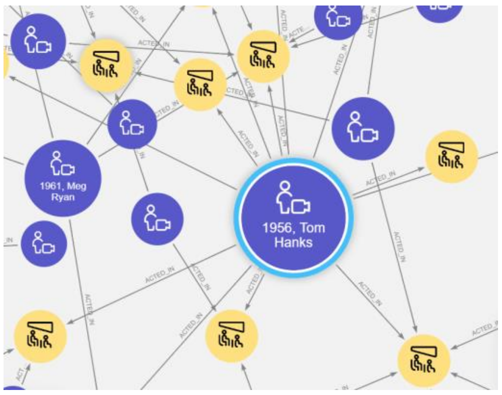
Minpoint: 0.1 | Size: 0.5x

Maxpoint: 5 | Size: 4x

Add rule-based styling

Movie 38

Person 102



Customizing relationships using default or Rule-based options

Similarly, as you did earlier, you can customize your relationships by:

- Selecting **Relationships** tab
- Select ACTED_IN relationship

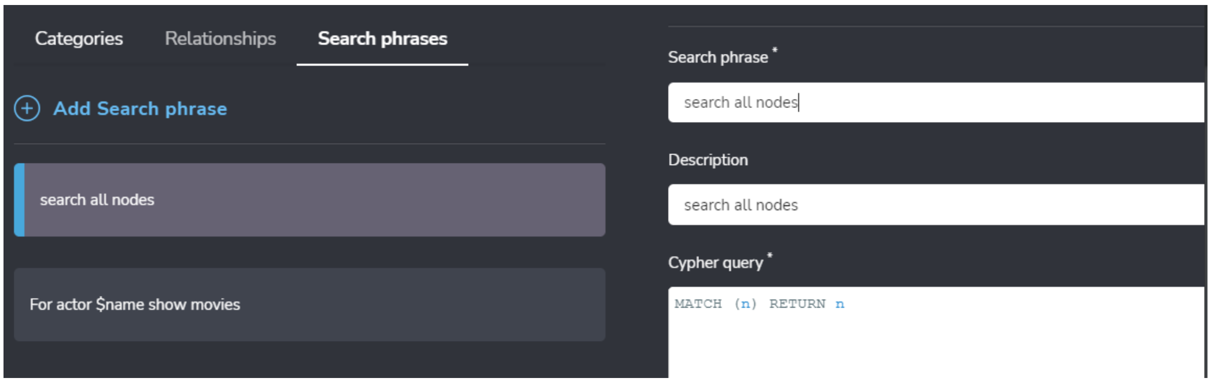
- Select **Default** options → unlike nodes, the relationships we have only color and size options. You can customize your relationships using color and size here
- Select **Rule-based** options → to customize your relationships using rule-based options

3.1.6. Creating search phrases

Search queries are queries that are used frequently. Search queries are meant to help analysts easily search for information in a Google-like search strategy by typing in search phrases.

Example: Creating search phrases – static phrases (e.g. search all nodes)

- Click on the perspective icon in your Neo4j Bloom browser
- Click on Search phrases
- Click on **(+) Add Search phrase** as illustrated below
- Specify or enter your search phrase and description as illustrated below
- Enter your Cypher query as illustrated below.

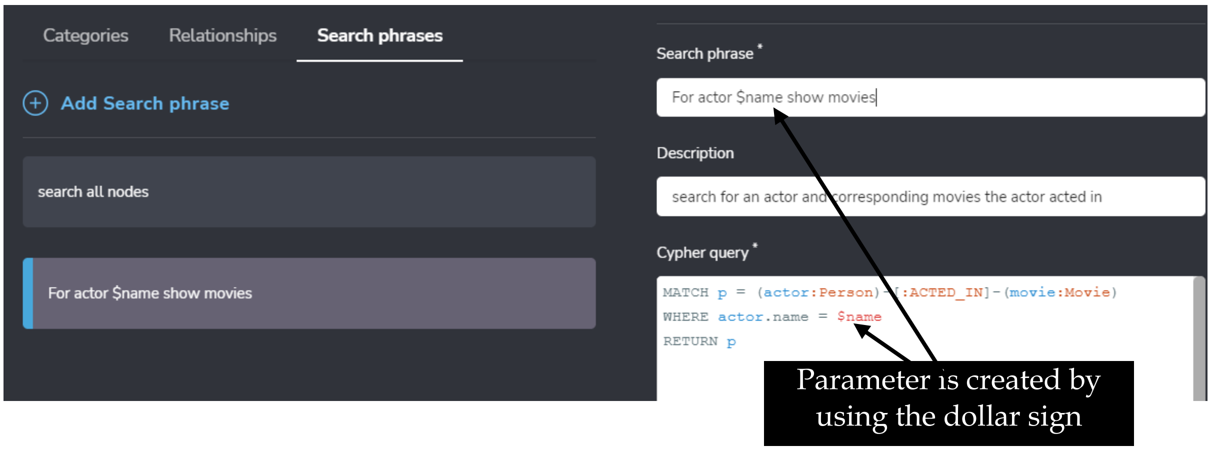


Search phrase *
search all nodes
Description
search all nodes
Cypher query *

```
MATCH (n) RETURN n
```

Example: Creating search phrases – with parameters

- Click on the perspective icon in your Neo4j Bloom browser
- Click on Search phrases
- Click on **(+) Add Search phrase** as illustrated below
- Enter your search phrase with a parameter (e.g. $param)
- Enter description of your search query
- Specify or select datatype and Label-key as illustrated below

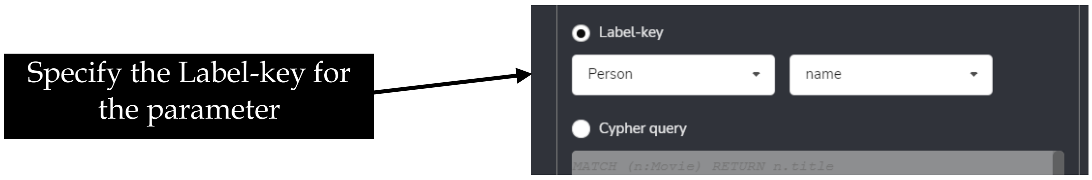

Categories
Relationships
Search phrases
Add Search phrase
search all nodes
For actor $name show movies
Parameter is created by using the dollar sign

Search phrase *: For actor $name show movies

Description: search for an actor and corresponding movies the actor acted in

Cypher query *:

```
MATCH p = (actor:Person)-[:ACTED_IN]-(movie:Movie)
WHERE actor.name = $name
RETURN p
```



[✓ Label-key]: Person, name

```
MATCH (n:Movie) RETURN n.title
```

Tasks to do:

- Use the search phrases you created