CSE1500 - WEB AND DATABASE TECHNOLOGY

DB LECTURE 6

# NOSQL / NEO4J

Alessandro Bozzon

cse1500-ewi@tudelft.nl

# AT THE END OF THIS LECTURE, YOU SHOULD BE ABLE TO….

- **Describe** and **design** data-driven applications using Neo4J

# NEO4J

# WHY?

Relational Databases

(incredibly!)

are not good in managing relationships!

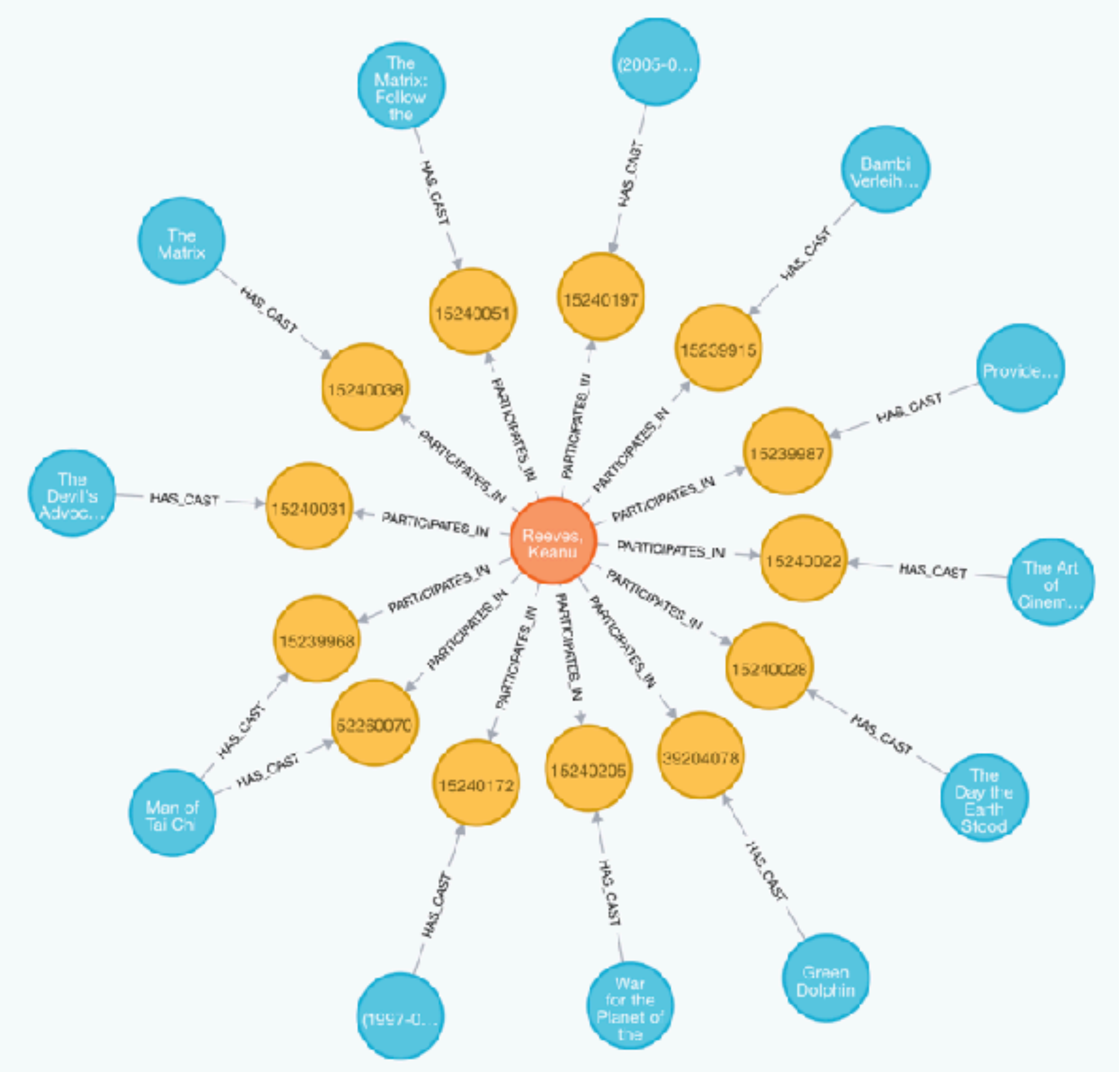
# THE NEO4J GRAPH DATABASE

- Open source: [http://neo4j.com](http://neo4j.com)
  - But with commercial – enterprise – edition (with caching, clustering, etc.)
  - Great documentation
- Implemented in Java, cross-platform
- Schemaless and “typeless”
  - Basic data types (e.g. string) , but no restrictions on entity and relationship types
- Declarative query language (**Cypher**)
  - On the path to standardisation http://www.opencypher.org/
- No sharding
- Master-Slave replication (but of entire graphs)
- **ACID** properties!

# PROS/CONS OF GRAPH DATABASES

- Break data into even smaller units that RDBMs

**Aggregates** **Relations** **Nodes**



- Not suitable for distribution, but ACID properties can be guaranteed (at a cost)
- Suitable for complex, semi-structured, highly connected data

# APPLICATIONS

- Connected Data
  - Social graphs, knowledge graphs, citations
- Routing, Dispatch, Location-Based services
  - Every location is a node, connections between locations can be a (weighted) edge
- Recommendation engines
  - Establish relationships between people and things

- Don’t use for massive data updates
  - It takes time to build and massively change graphs

Walmart

TOMTOM

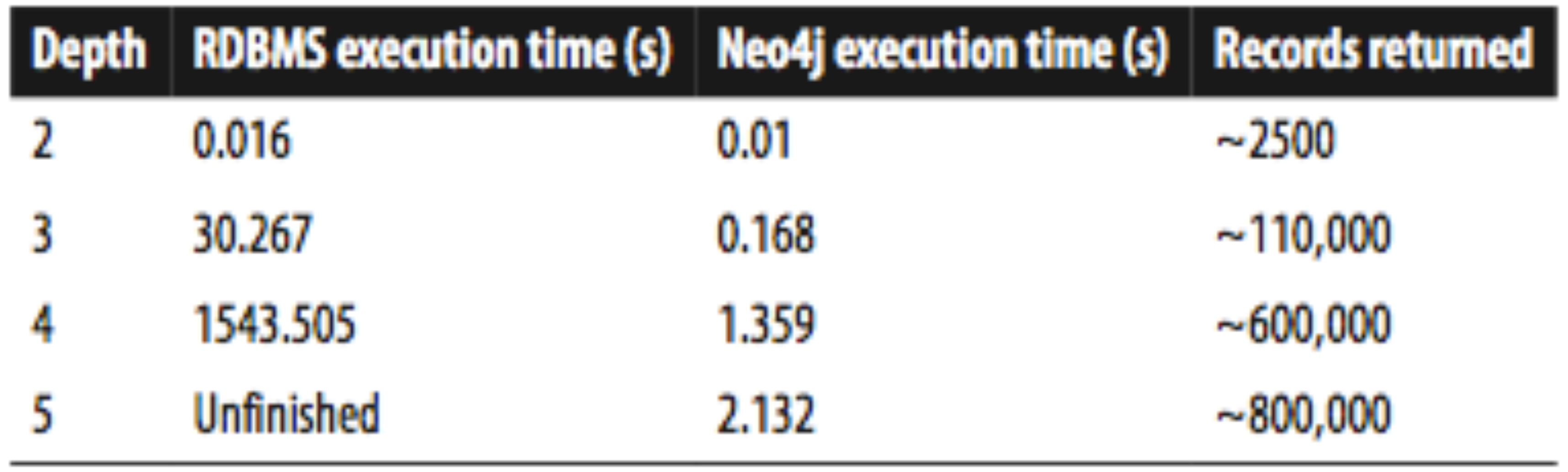
Linkedin 领英

# GRAPH VS. RELATIONAL DATABASES

- RDBMs and NOSQL DBs lack relationships
  - JOINS and aggregates are sub-optimal solutions
  - Relationships (through reference or embedding) are not reflective
- Graph Databases embrace relationships
  - Traversal operation are highly efficient

Friends of friends of my friends?

THIS IS WITHOUT RECURSIVE COMMON TABLE EXPRESSIONS



| Depth | RDBMS execution time (s) | Neo4j execution time (s) | Records returned |
|---|---|---|---|
| 2 | 0.016 | 0.01 | ~2500 |
| 3 | 30.267 | 0.168 | ~110,000 |
| 4 | 1543.505 | 1.359 | ~600,000 |
| 5 | Unfinished | 2.132 | ~800,000 |

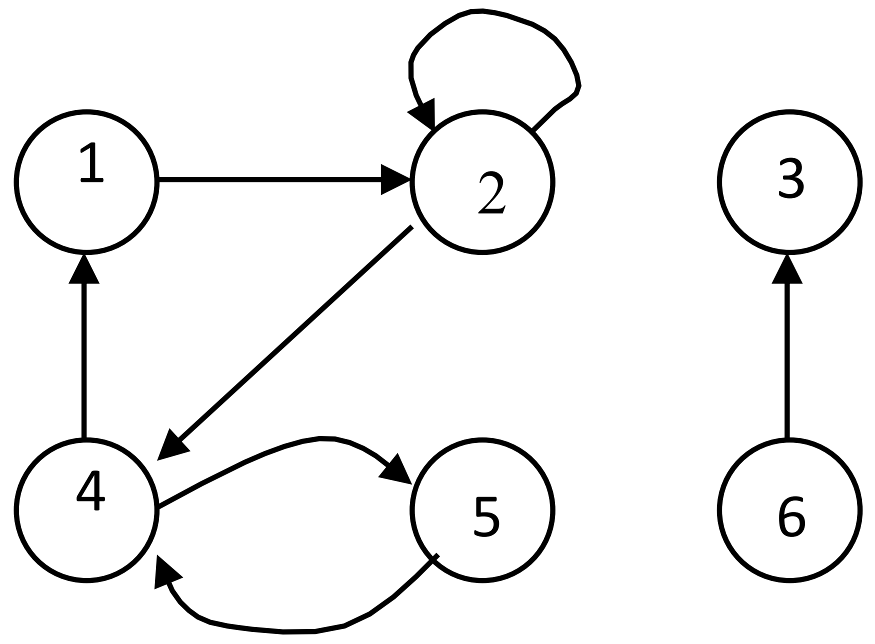
- In a depth two (friend-of-friend), both RDBMs and Neo4J perform well enough
  - But when we do the depth three it clear that RDBMs can no longer keep up

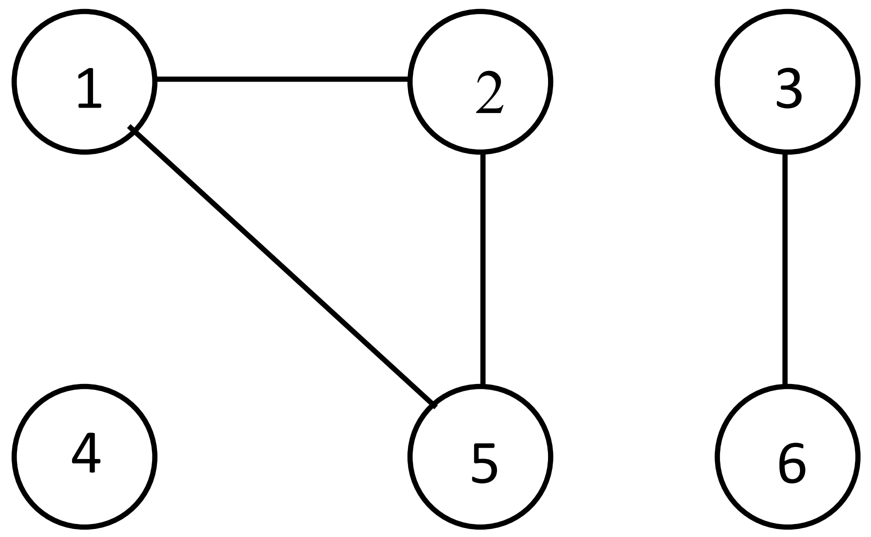
http://neo4j.com/news/how-much-faster-is-a-graph-database-really/

# NEO4J DATA MODEL

# WHAT IS A GRAPH?

- Informally, a graph is a set of nodes joined by a set of directed or undirected edges
- A subgraph of a graph G is a graph fully contained by G

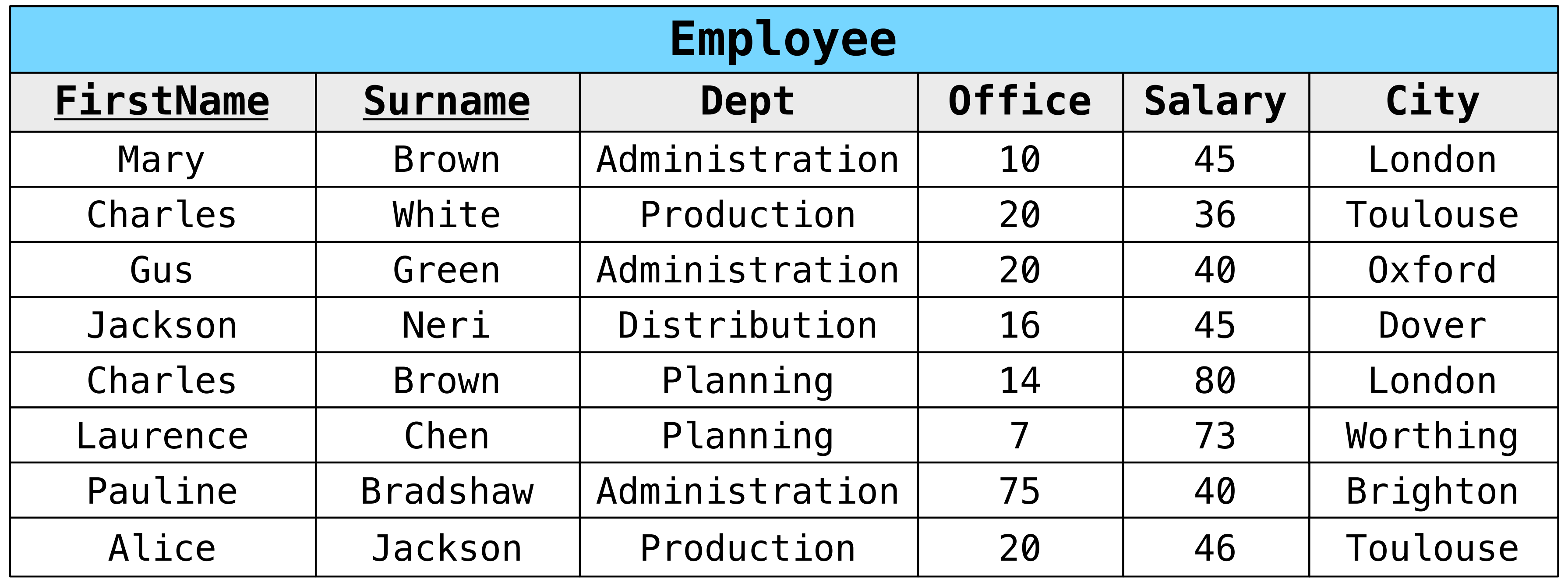




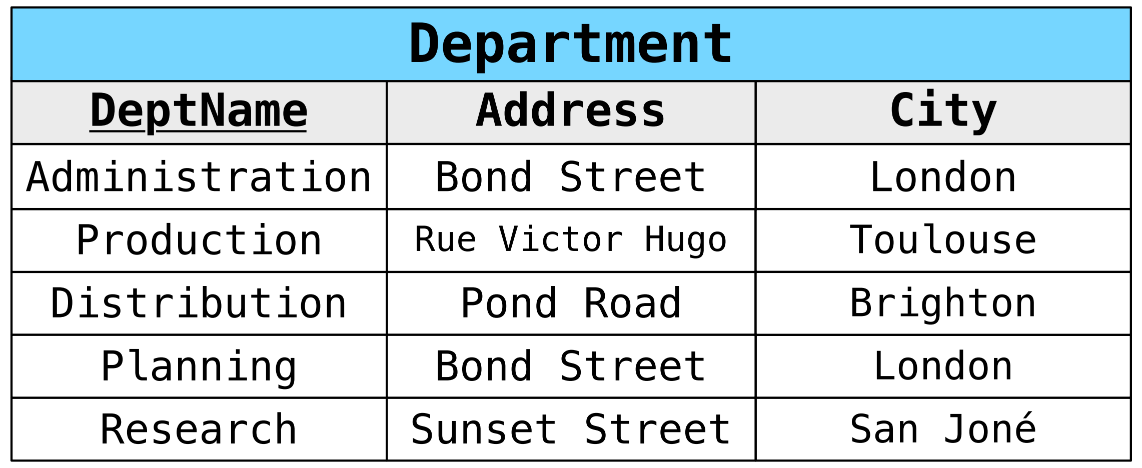
- Almost anything can be modelled as a graph

# RELATIONAL DATABASE REPRESENTATION

## Relational Model

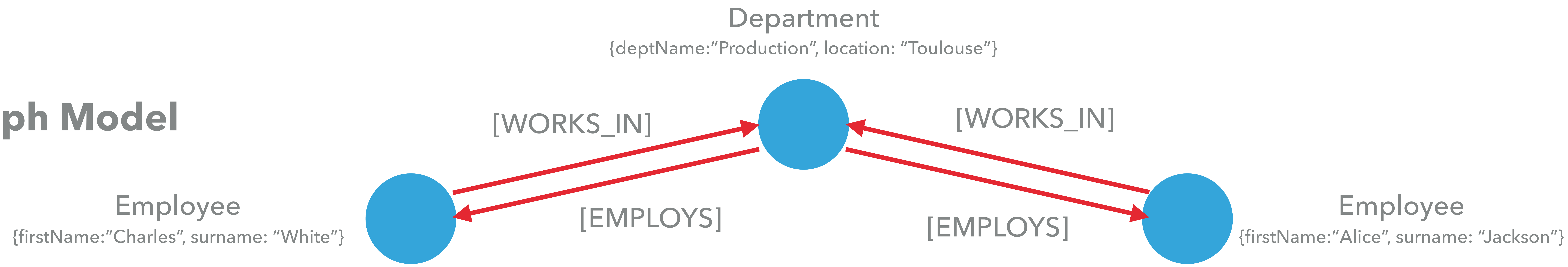


| Employee | | | | | |
|---|---|---|---|---|---|
| **FirstName** | **Surname** | **Dept** | **Office** | **Salary** | **City** |
| Mary | Brown | Administration | 10 | 45 | London |
| Charles | White | Production | 20 | 36 | Toulouse |
| Gus | Green | Administration | 20 | 40 | Oxford |
| Jackson | Neri | Distribution | 16 | 45 | Dover |
| Charles | Brown | Planning | 14 | 80 | London |
| Laurence | Chen | Planning | 7 | 73 | Worthing |
| Pauline | Bradshaw | Administration | 75 | 40 | Brighton |
| Alice | Jackson | Production | 20 | 46 | Toulouse |



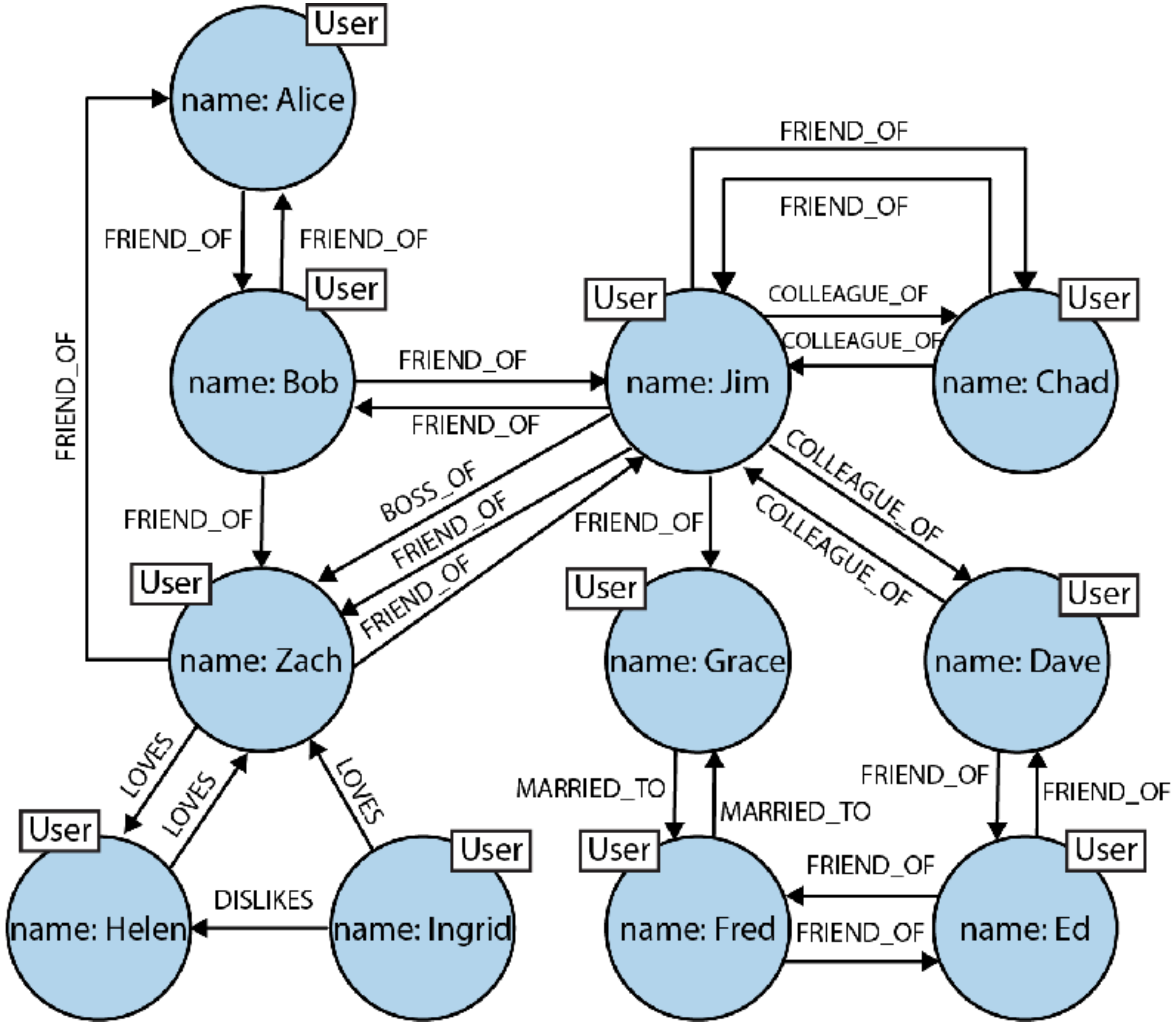
| Department | | |
|---|---|---|
| **DeptName** | **Address** | **City** |
| Administration | Bond Street | London |
| Production | Rue Victor Hugo | Toulouse |
| Distribution | Pond Road | Brighton |
| Planning | Bond Street | London |
| Research | Sunset Street | San Joné |

## Graph Model


Department
{deptName:"Production", location: "Toulouse"}
[WORKS_IN]
[WORKS_IN]
Employee
{firstName:"Charles", surname: "White"}
[EMPLOYS]
[EMPLOYS]
Employee
{firstName:"Alice", surname: "Jackson"}

# NEO4J DATA MODEL: LABELED PROPERTY GRAPH

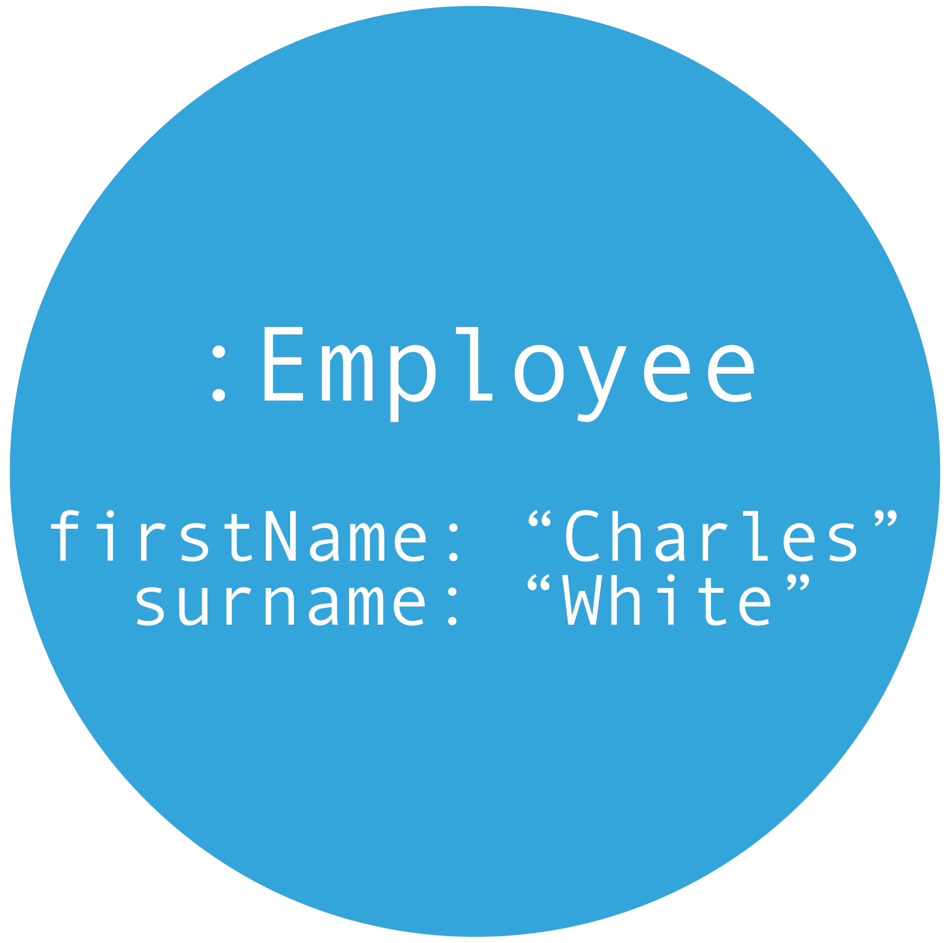
- **Labeled Property Graph**
- **Nodes** have *labels* (types)
- **Nodes** can have *properties* (name-value pairs)
- **Edges** have a start node, an end node, and a direction
- **Edges** can also have *name* and *properties*



Credits: Graph Databases, 2nd Edition

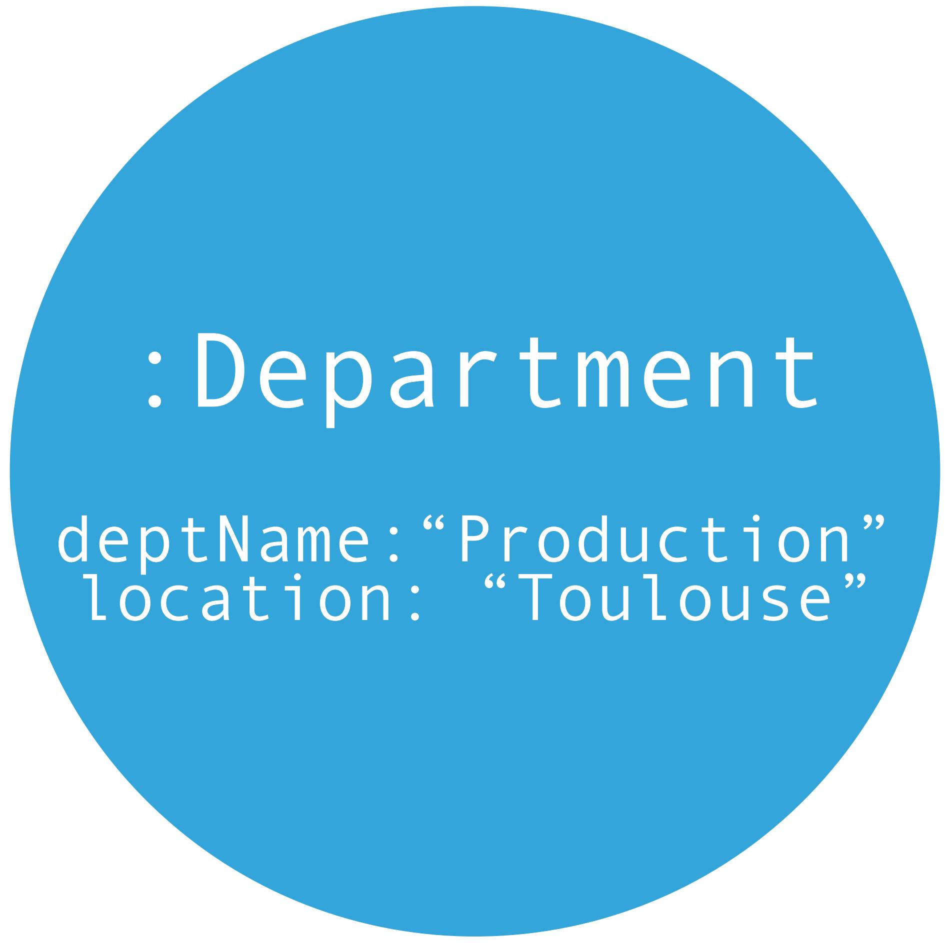
# NODES IN NEO4J

- Represent **entities** with a unique conceptual identity
  - Have unique (internal) identifier
  - Can have **types**, defined by one or more **labels**
- **Labels**
  - Nodes having the same label belong to the same set
  - Can be added and removed at runtime
  - Naming convention: Camel case, beginning with an upper-case character
- **Properties**
  - **name**:**value** pairs (no lists or array – deprecated)
  - **name** is a string
  - Standard data types (Integer, Float, String, Boolean, spatial Point, DateTime)
  - Lower camel case, beginning with a lower-case character



:Employee

firstName: “Charles”
surname: “White”

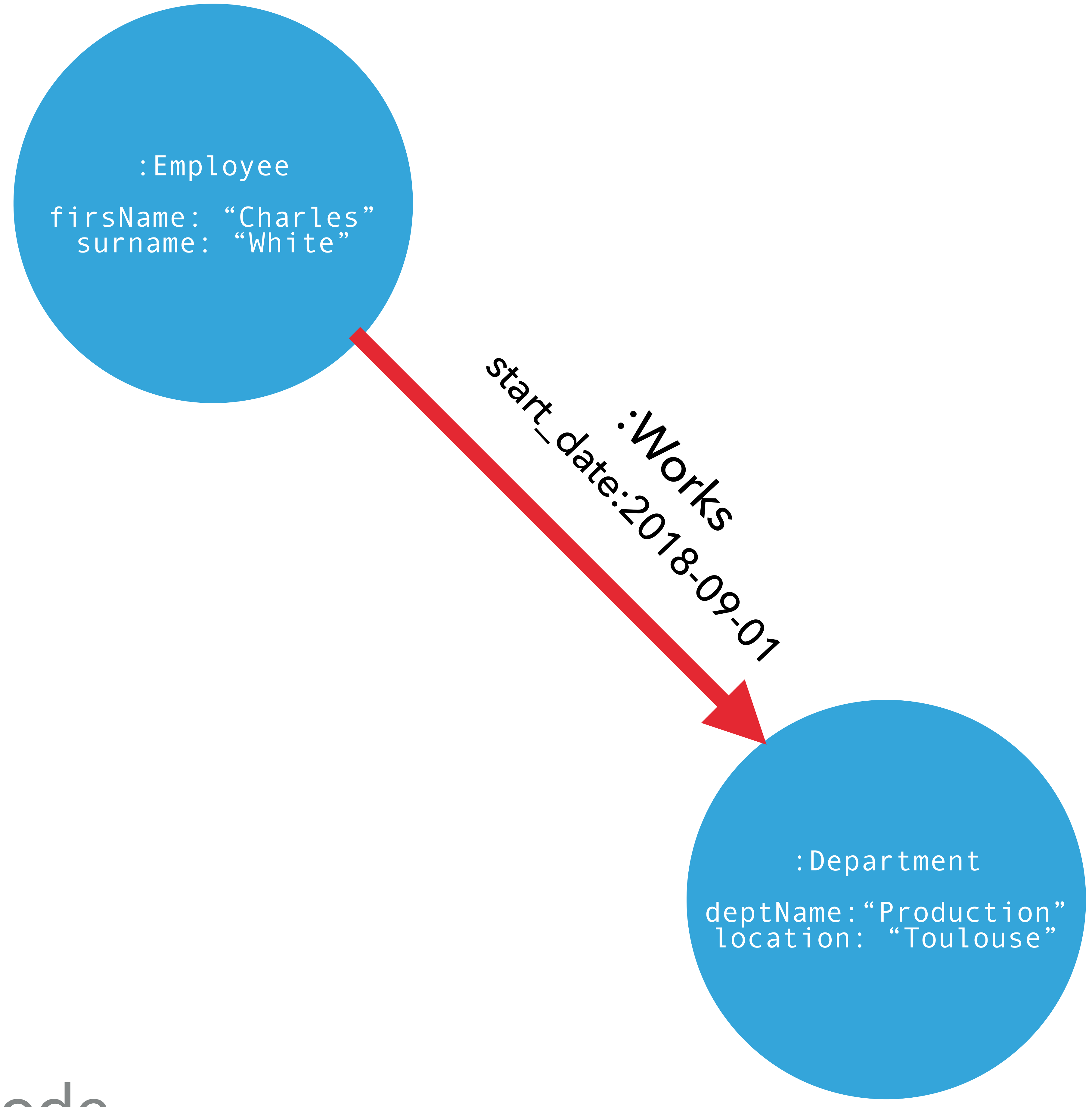


:Department

deptName:“Production”
location: “Toulouse”

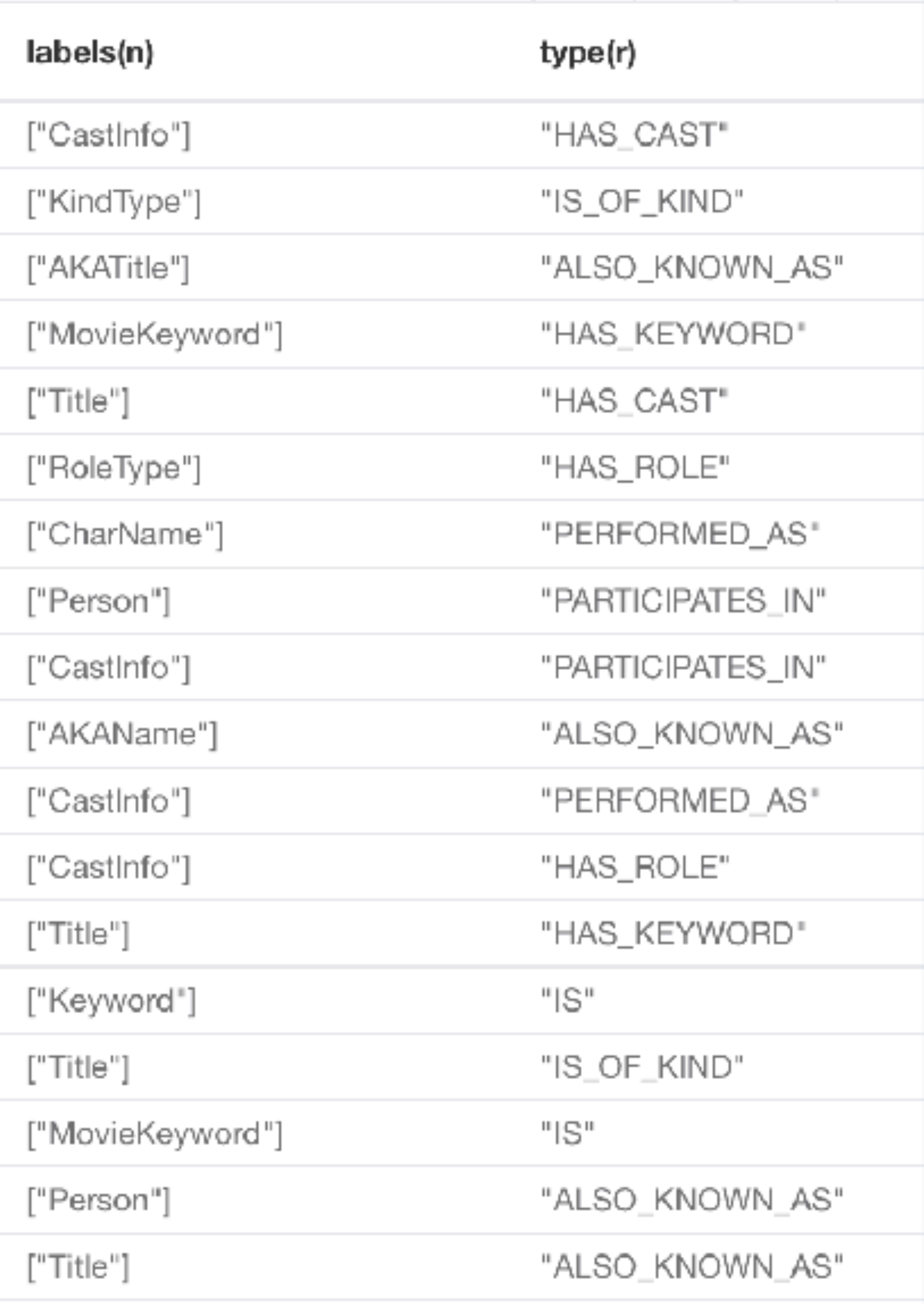
# RELATIONSHIPS

- It connects *source node* and an *target node*
  - Directions are **not** binding at query time
  - Loops are allowed
- Relationship **Type**
  - The name of the relationship
  - Upper case, using underscore to separate words
- **Properties**
  - name:value pairs - as for nodes
- Built in *"referential integrity"*
  - An existing relationship never points to a non-existing node
  - You cannot delete a node without deleting its associated relationships

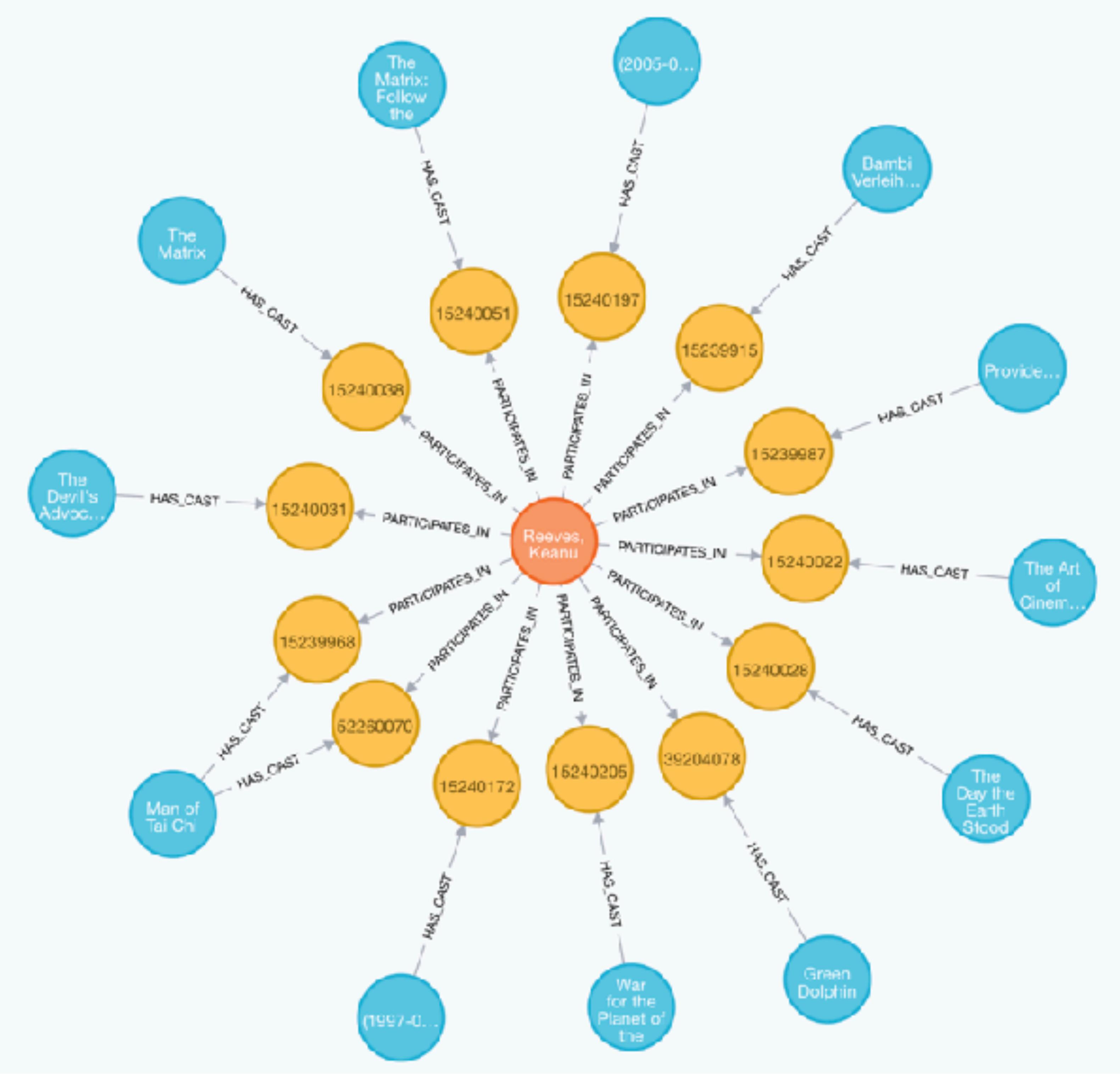


# EXAMPLE DATABASE: IMDB

- Node labels
  - One for each IMDB table
  - Properties as in the matching IMDB tables
- Relationship types
  - One for each referential integrity constraint in the relational version



| labels(n) | type(r) |
|---|---|
| ["CastInfo"] | "HAS_CAST" |
| ["KindType"] | "IS_OF_KIND" |
| ["AKATitle"] | "ALSO_KNOWN_AS" |
| ["MovieKeyword"] | "HAS_KEYWORD" |
| ["Title"] | "HAS_CAST" |
| ["RoleType"] | "HAS_ROLE" |
| ["CharName"] | "PERFORMED_AS" |
| ["Person"] | "PARTICIPATES_IN" |
| ["CastInfo"] | "PARTICIPATES_IN" |
| ["AKAName"] | "ALSO_KNOWN_AS" |
| ["CastInfo"] | "PERFORMED_AS" |
| ["CastInfo"] | "HAS_ROLE" |
| ["Title"] | "HAS_KEYWORD" |
| ["Keyword"] | "IS" |
| ["Title"] | "IS_OF_KIND" |
| ["MovieKeyword"] | "IS" |
| ["Person"] | "ALSO_KNOWN_AS" |
| ["Title"] | "ALSO_KNOWN_AS" |



# QUERYING NEO4J

# CYPHER

- **Declarative** graph query language
- Allows for expressive and efficient querying and updates
  - Inspired by SQL (query clauses) and SPARQL (pattern matching)
- Many features stem from improving on SQL pain points
  - e.g. `JOIN`s, `GROUP BY`

```
MATCH (t:Title{
           title: 'The Matrix',
           production_year: 1999}
       )-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)

RETURN distinct p.name
```

Query: find the name of all the actors that played a part in the movie “The Matrix” produced in 1999

# QUERY CLAUSES

- `MATCH`
  - Specifies **graph patterns** to be searched for
- `WHERE`
  - **Filtering** constraints on nodes and relationships
- `RETURN`
  - Defines **returned** data
- `ORDER BY`
  - Describes how results should be **ordered**
- `UNION`
  - Merges results from two or more queries
- `WITH`
  - Chains subsequent query parts and forwards results from one to the next

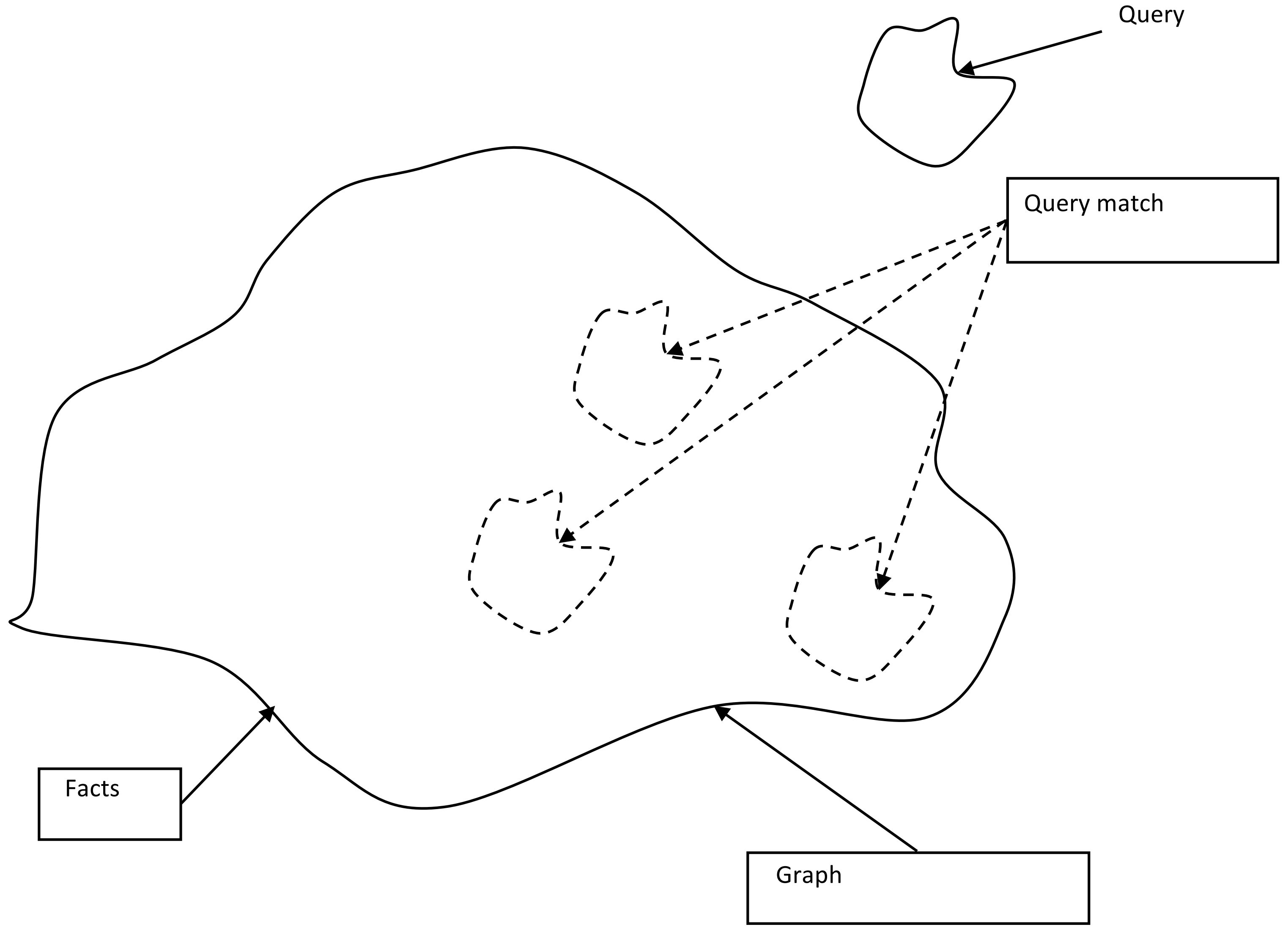
# CYPHER PRINCIPLES

- Declarative
- Specification by example
- Patterns specified with ASCII-Art inspired syntax
  - Circles `()` for nodes
  - Arrows `<–`, `--`, `–>` and `[]` for relationships

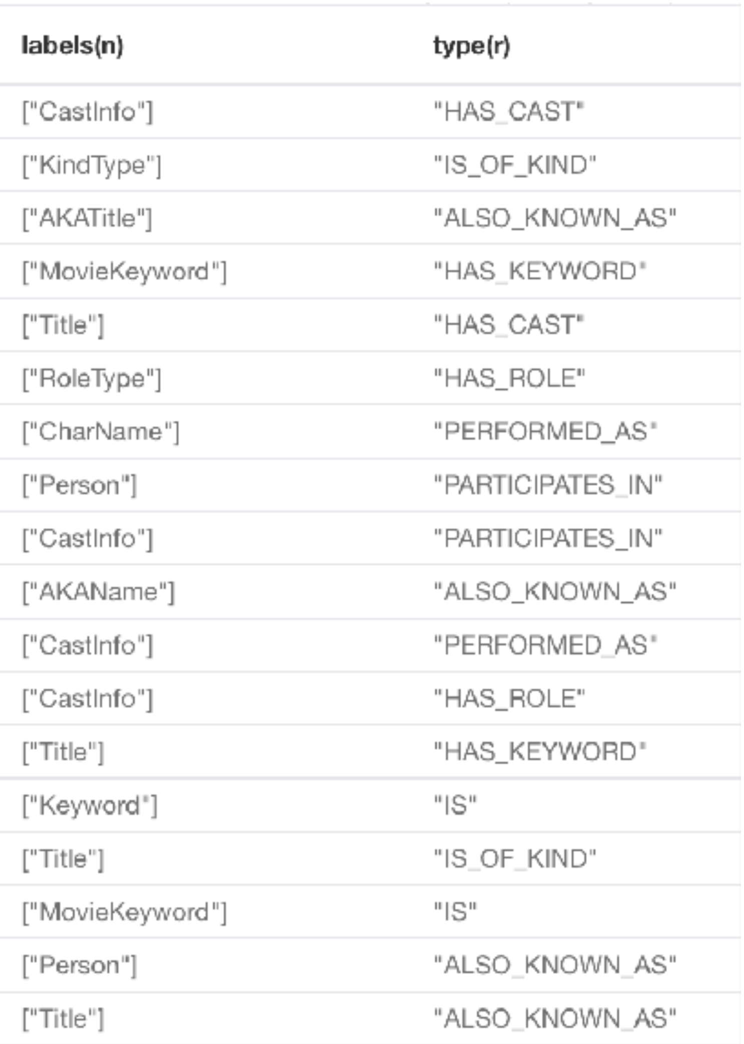
```
MATCH (t:Title{
           title: 'The Matrix',
           production_year: 1999}
       )-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)

RETURN distinct p.name
```

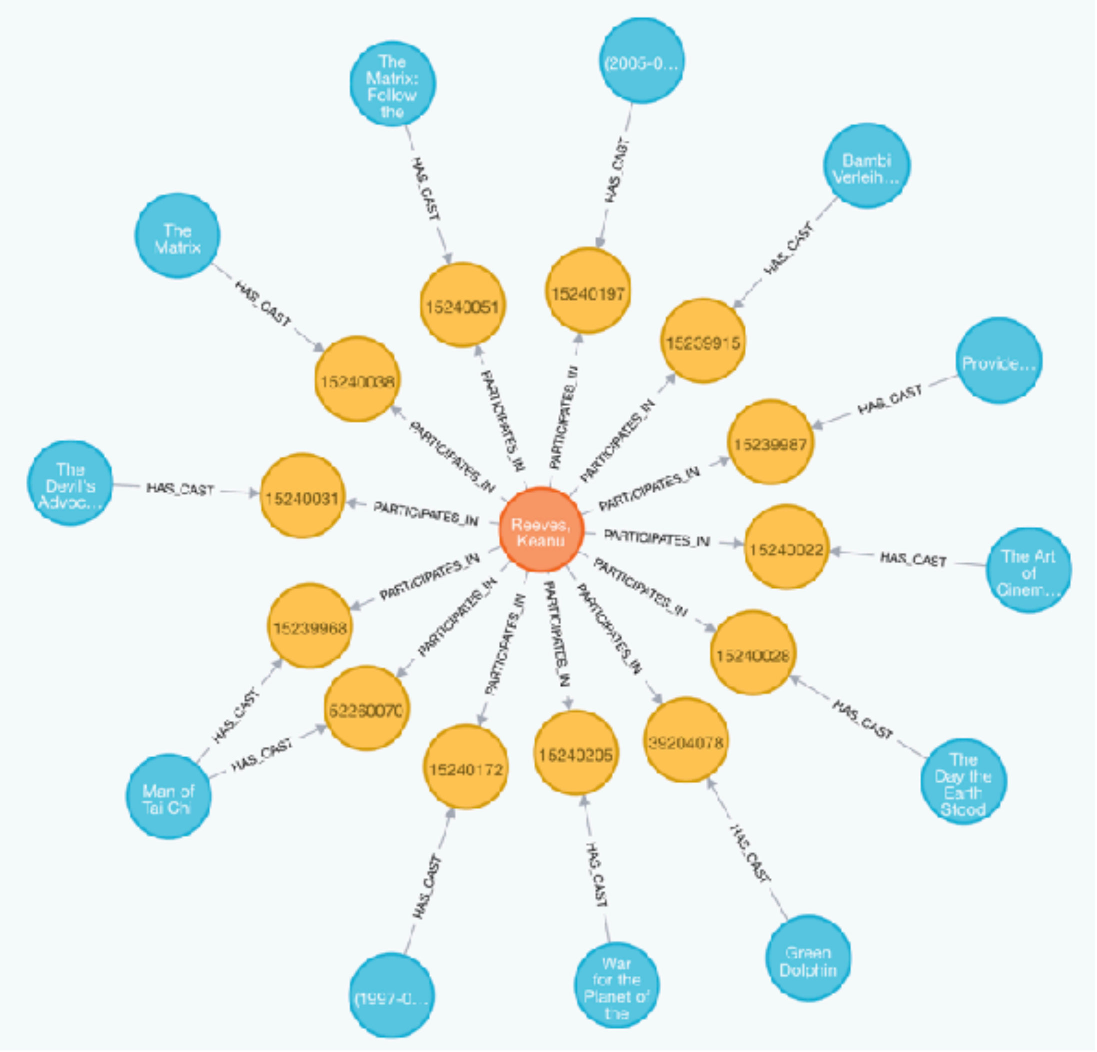
# MATCH CLAUSE



- Search for sub-graphs of the data graph that match **all** the **patterns** in the query
  - Patterns can occur many times in the graph
- The query result are set of **variable bindings**
  - each variable has to be bound to a node/relationship
- Queries can return:
  - a **graph** (node and relationship instances)
  - a **table** (node and relationship properties)
- Results are unordered



| labels(n) | type(r) |
|---|---|
| ["CastInfo"] | "HAS_CAST" |
| ["KindType"] | "IS_OF_KIND" |
| ["AKATitle"] | "ALSO_KNOWN_AS" |
| ["MovieKeyword"] | "HAS_KEYWORD" |
| ["Title"] | "HAS_CAST" |
| ["RoleType"] | "HAS_ROLE" |
| ["CharName"] | "PERFORMED_AS" |
| ["Person"] | "PARTICIPATES_IN" |
| ["CastInfo"] | "PARTICIPATES_IN" |
| ["AKAName"] | "ALSO_KNOWN_AS" |
| ["CastInfo"] | "PERFORMED_AS" |
| ["CastInfo"] | "HAS_ROLE" |
| ["Title"] | "HAS_KEYWORD" |
| ["Keyword"] | "IS" |
| ["Title"] | "IS_OF_KIND" |
| ["MovieKeyword"] | "IS" |
| ["Person"] | "ALSO_KNOWN_AS" |
| ["Title"] | "ALSO_KNOWN_AS" |



# NODE PATTERN

```
(variable:Label:..:Label{key: expression, …, key: expression})
```

- Matches nodes
- **Variable**: used to refer/access nodes
- **Labels**: all node labels to be matched
- **Properties**: properties of the nodes to be matched
  - Order is not important

# EXAMPLES

```
(t)
```

- Any node, referred by the variable `t`

```
(t:Title)
```

- Nodes with label `Title`

```
(t:Title:Person)
```

- Nodes with labels `Title` **and** `Person`

```
(t:Title{title:'The Matrix',production_year: 1999})
```

- Nodes with label `Title` having properties with the specified values

# PATH PATTERNS

```
()<-|-[variable:type|…|type variableLenght {key: expression, …, key: expression}]-|->()
```

- Describe multiple nodes and one or more relationships among them
- A series of connected nodes is called a **path**
  - A traversal of part of the graph
  - Describes a single path, not a general sub-graph
  - Used as part of a query to specify patterns
- **Variable**: used to refer/access relationships
- **Type**: all relationship types to be matched
- **VariableLenght**: describe paths of arbitrary lengths (not just one relationship)
- **Properties**: properties of the relationships to be matched

# EXAMPLES PATH PATTERNS /1

```
(t)-[r:HAS_CAST]-(c)
```

- Matches relationships of type `HAS_CAST` between all nodes

```
(t)->(c)
```

- Matches all relationships from `t` (any node) to `c` (any node)

```
(t)--(c)
```

- Matches relationships in any direction between `t` to `c`

```
(t:Title)—>(c)
```

- All relationships from nodes with label `Title`

# EXAMPLES PATH PATTERNS /2

```
(t)<-[HAS_CAST]-(c)
```

- Relationships of type HAS_CAST from c to t

```
(t)-[HAS_CAST|HAS_KEYWORD]->(c)
```

- Matches relationships of type HAS_CAST or HAS_KEYWORD from t to c

```
(t)-[r]-(c)
```

- Binds the relationships to the variable r

```
(t)-[*1..5]-(c)
```

- Variable length path, from 1 to 5 relationships, between t and c

# EXAMPLES PATH PATTERNS /3

```
(t)-[*]->(c)
```

- Variable path of any number of 5 relationships, from `t` to `c`

```
(t)-[HAS_CAST]-(c {movie_id:'2543774'})
```

- A relationship of type `HAS_CAST` from a node `t` to a node `c` having declared property

```
shortestPath((p1:Person)-[*..6]-((p2:Person))
```

- Find a single shortest path between any two nodes `Person`

```
size((t)-->()-->())
```

- Count the paths matching the pattern

# RETURN CLAUSE

- Defines what to include in the query result set
- Projection of node and relationship variables
  - Properties accessed via dot notation
    - Missing properties are treated as `null`
  - `*` returns all the variables
  - `AS` allows to explicitly (re)name results
- Literals, predicates, properties, functions can also be returned
- `DISTINCT`
  - retrieves only unique rows depending on the columns that have been selected to output

# EXAMPLES RETURN /1

Returns distinct people name

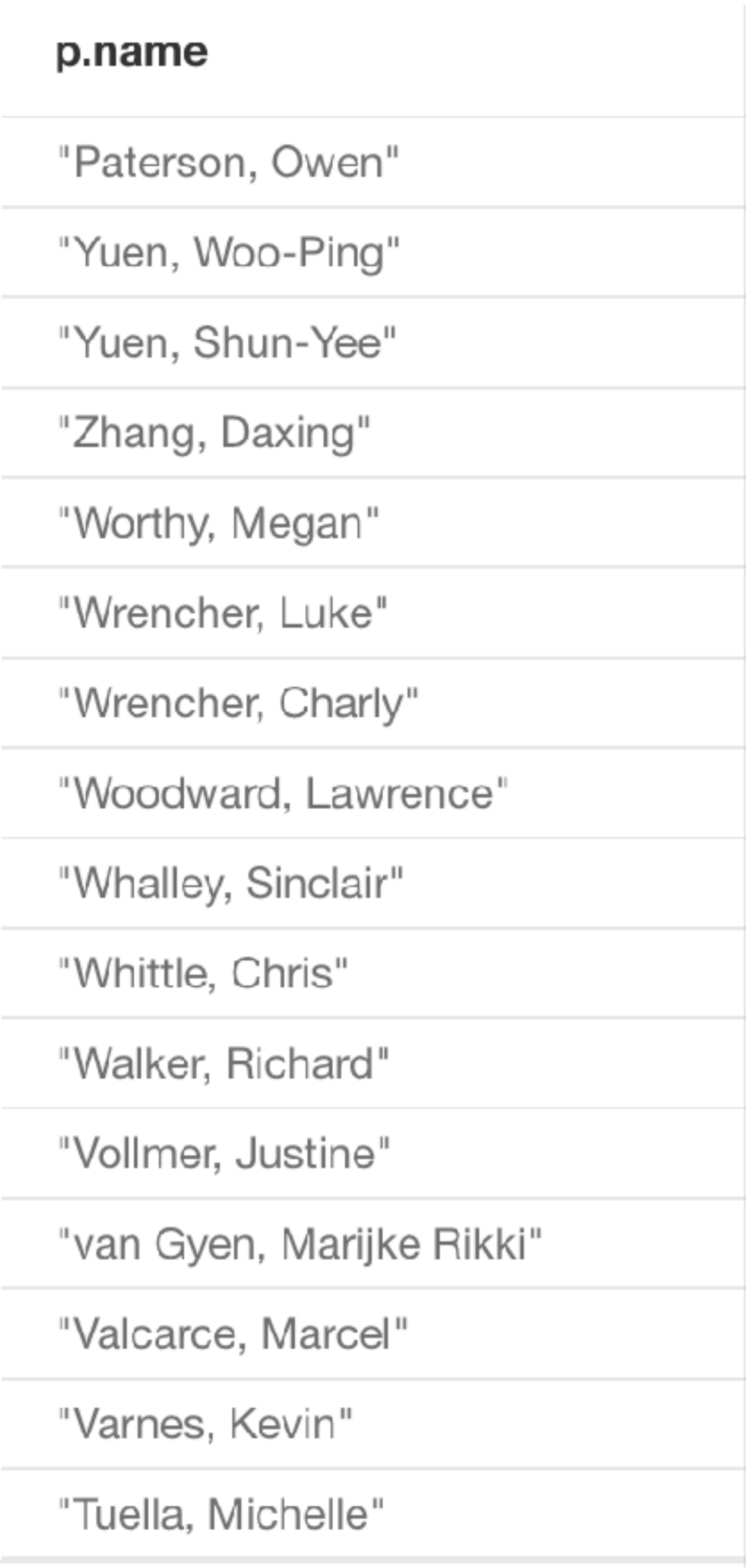
```
MATCH (t:Title{
           title: 'The Matrix',
           production_year: 1999}
      )-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
RETURN distinct p.name
```

Returns distinct people name, a `true/false/null` value if they are female, and a constant literal

```
RETURN distinct p.name, p.gender = ‘f’, ‘A Literal’
```

Renaming and expressions

```
RETURN t.production_year-2000 AS Millenium_Age
```

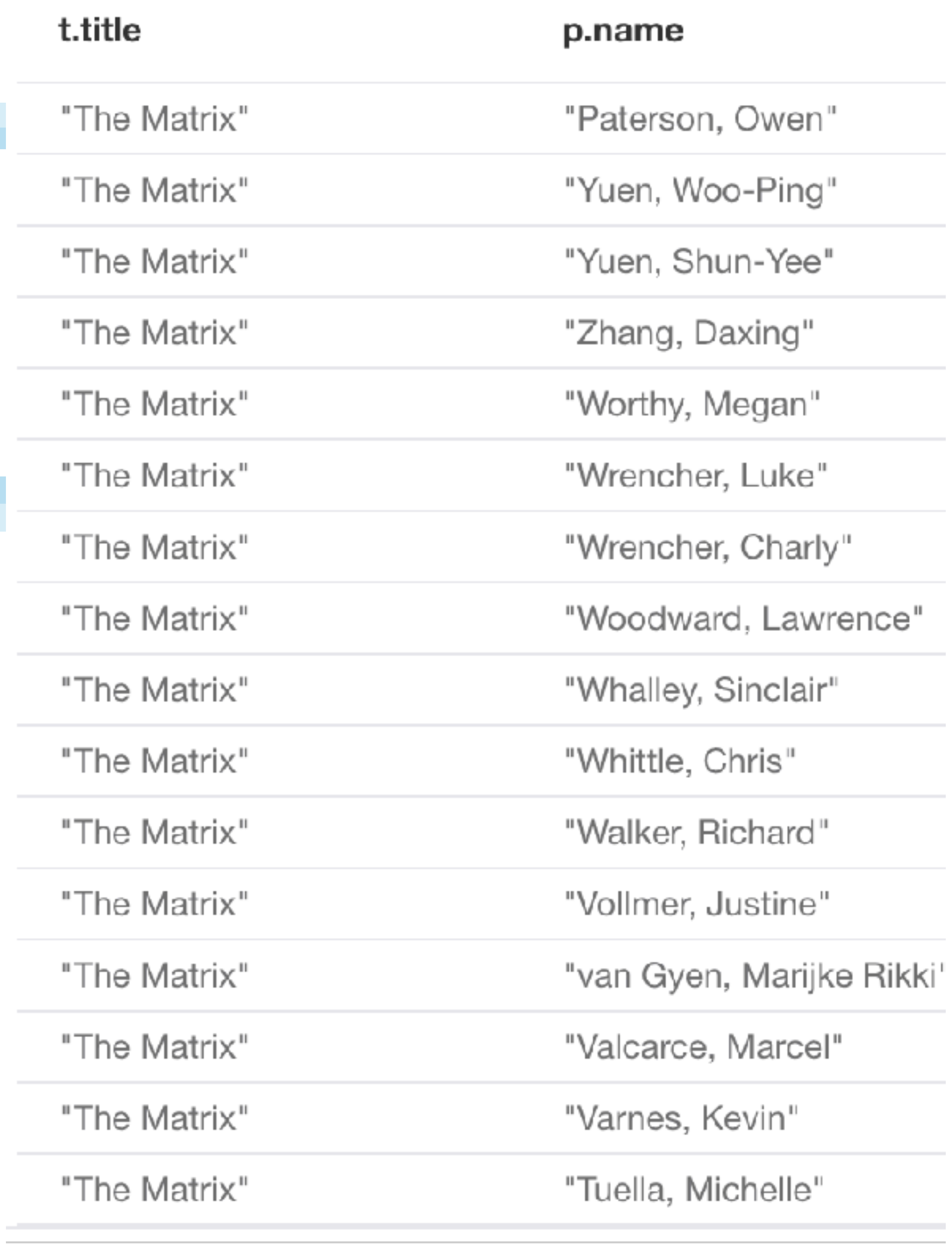


| p.name |
|---|
| "Paterson, Owen" |
| "Yuen, Woo-Ping" |
| "Yuen, Shun-Yee" |
| "Zhang, Daxing" |
| "Worthy, Megan" |
| "Wrencher, Luke" |
| "Wrencher, Charly" |
| "Woodward, Lawrence" |
| "Whalley, Sinclair" |
| "Whittle, Chris" |
| "Walker, Richard" |
| "Vollmer, Justine" |
| "van Gyen, Marijke Rikki" |
| "Valcarce, Marcel" |
| "Varnes, Kevin" |
| "Tuella, Michelle" |

# EXAMPLES RETURN /2

```
MATCH (t:Title{
           title: 'The Matrix',
           production_year: 1999}
       )-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
RETURN distinct t.title, p.name
```

Returns distinct movie name and people name

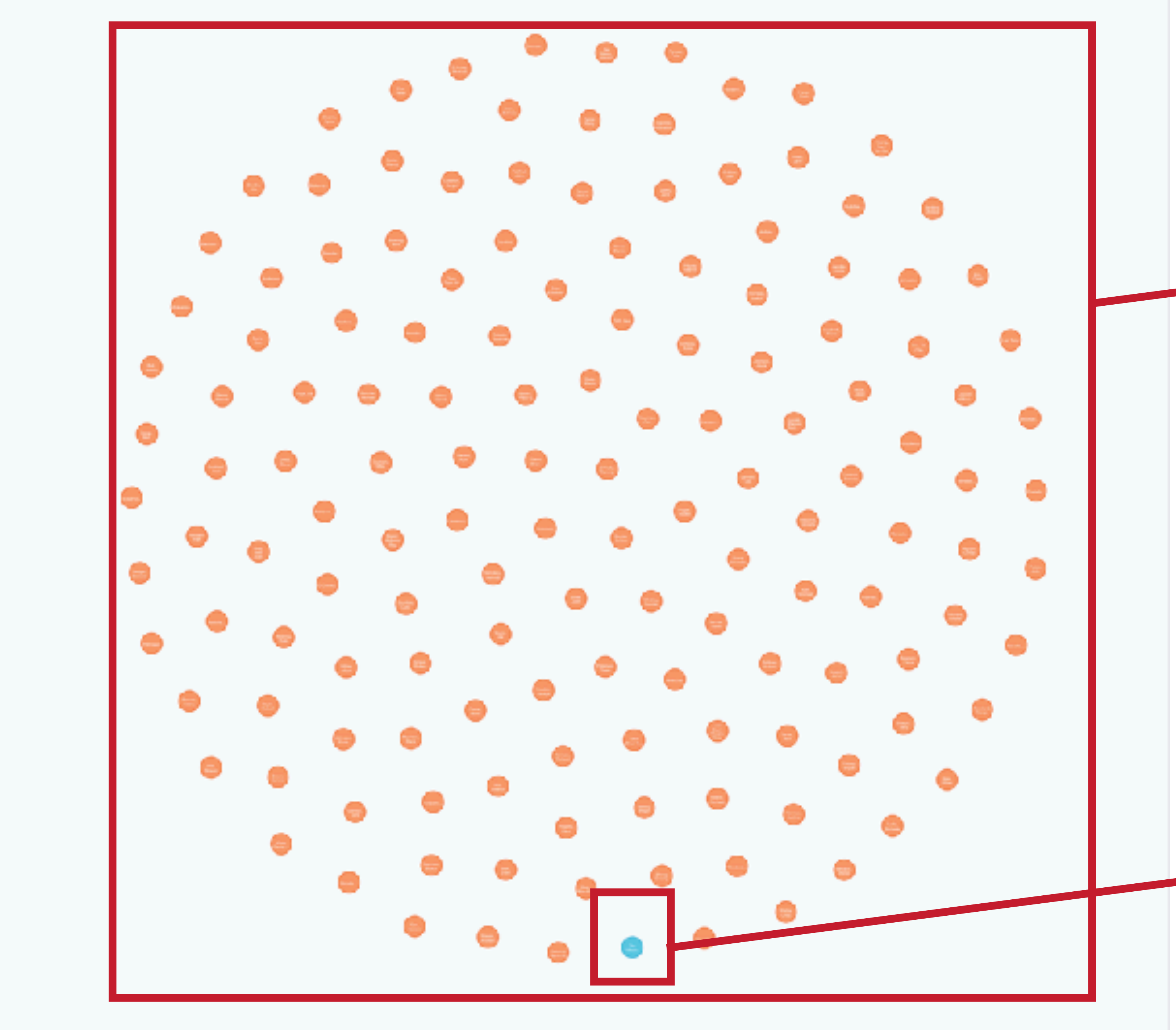


| t.title | p.name |
|---|---|
| "The Matrix" | "Paterson, Owen" |
| "The Matrix" | "Yuen, Woo-Ping" |
| "The Matrix" | "Yuen, Shun-Yee" |
| "The Matrix" | "Zhang, Daxing" |
| "The Matrix" | "Worthy, Megan" |
| "The Matrix" | "Wrencher, Luke" |
| "The Matrix" | "Wrencher, Charly" |
| "The Matrix" | "Woodward, Lawrence" |
| "The Matrix" | "Whalley, Sinclair" |
| "The Matrix" | "Whittle, Chris" |
| "The Matrix" | "Walker, Richard" |
| "The Matrix" | "Vollmer, Justine" |
| "The Matrix" | "van Gyen, Marijke Rikki' |
| "The Matrix" | "Valcarce, Marcel" |
| "The Matrix" | "Varnes, Kevin" |
| "The Matrix" | "Tuella, Michelle" |

# EXAMPLES RETURN /3

```
MATCH (t:Title{
           title: 'The Matrix',
           production_year: 1999}
       )-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
RETURN t,p
```

Returns a collection of nodes



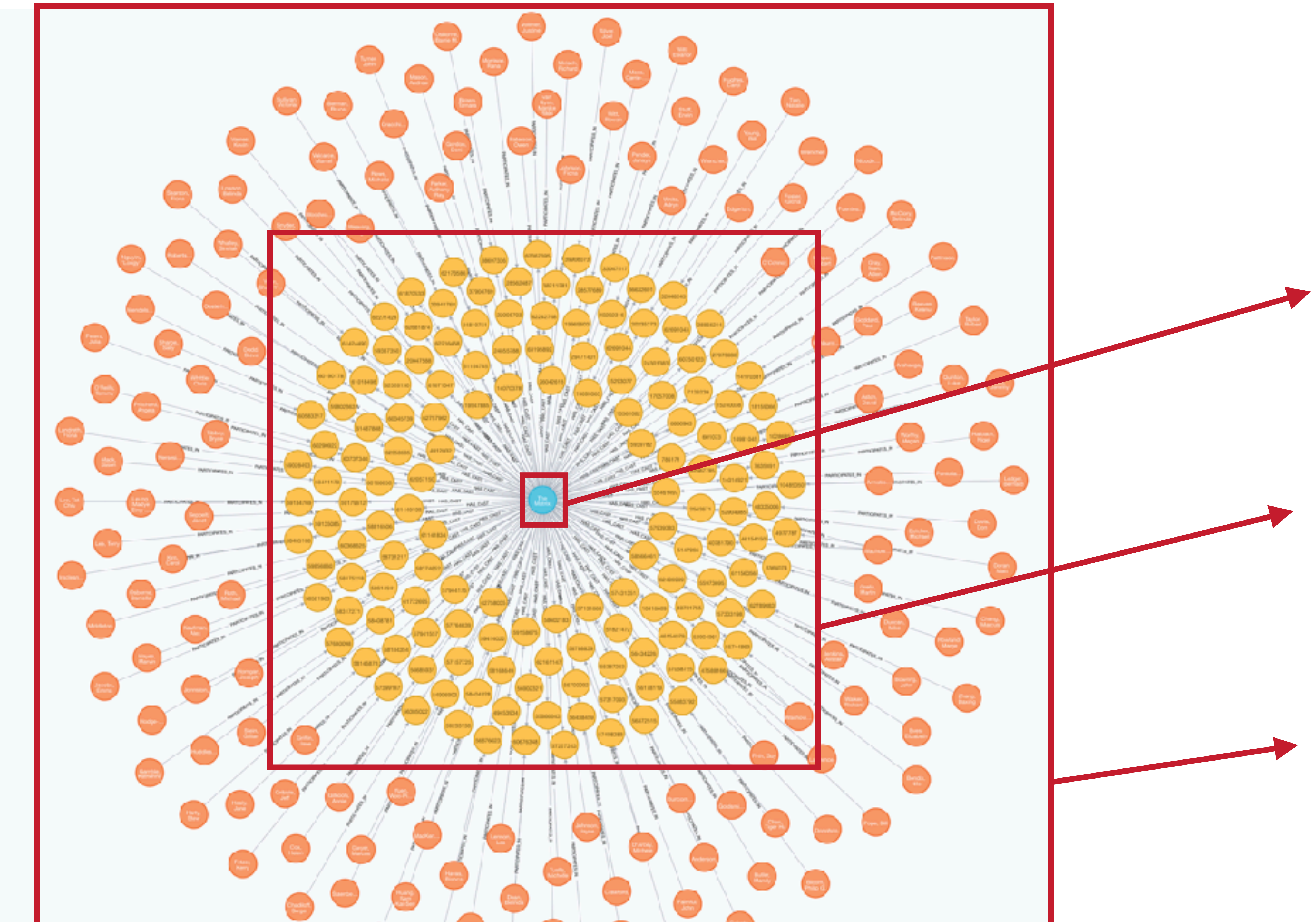
Person

The Matrix

# EXAMPLES RETURN /4

Returns a graph

```
MATCH (t:Title{
           title: 'The Matrix',
           production_year: 1999}
      )-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
RETURN t,r,c,s,p
```



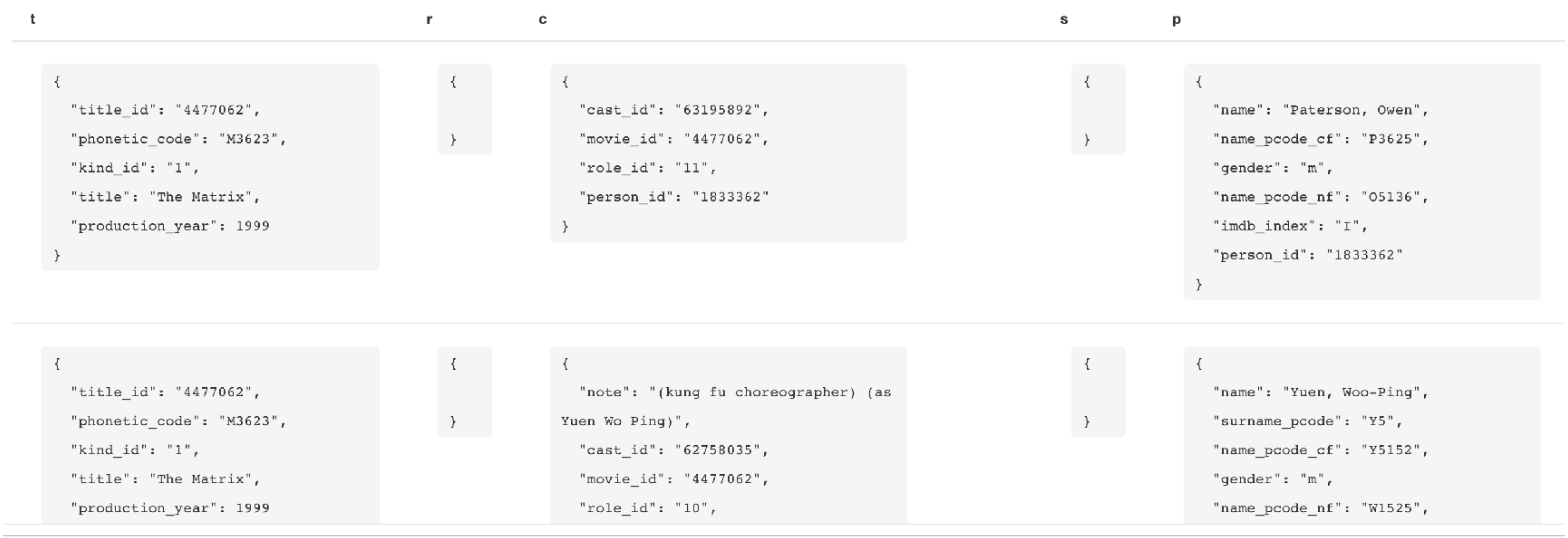
The Matrix

CastInfo

Person

# EXAMPLES RETURN /4

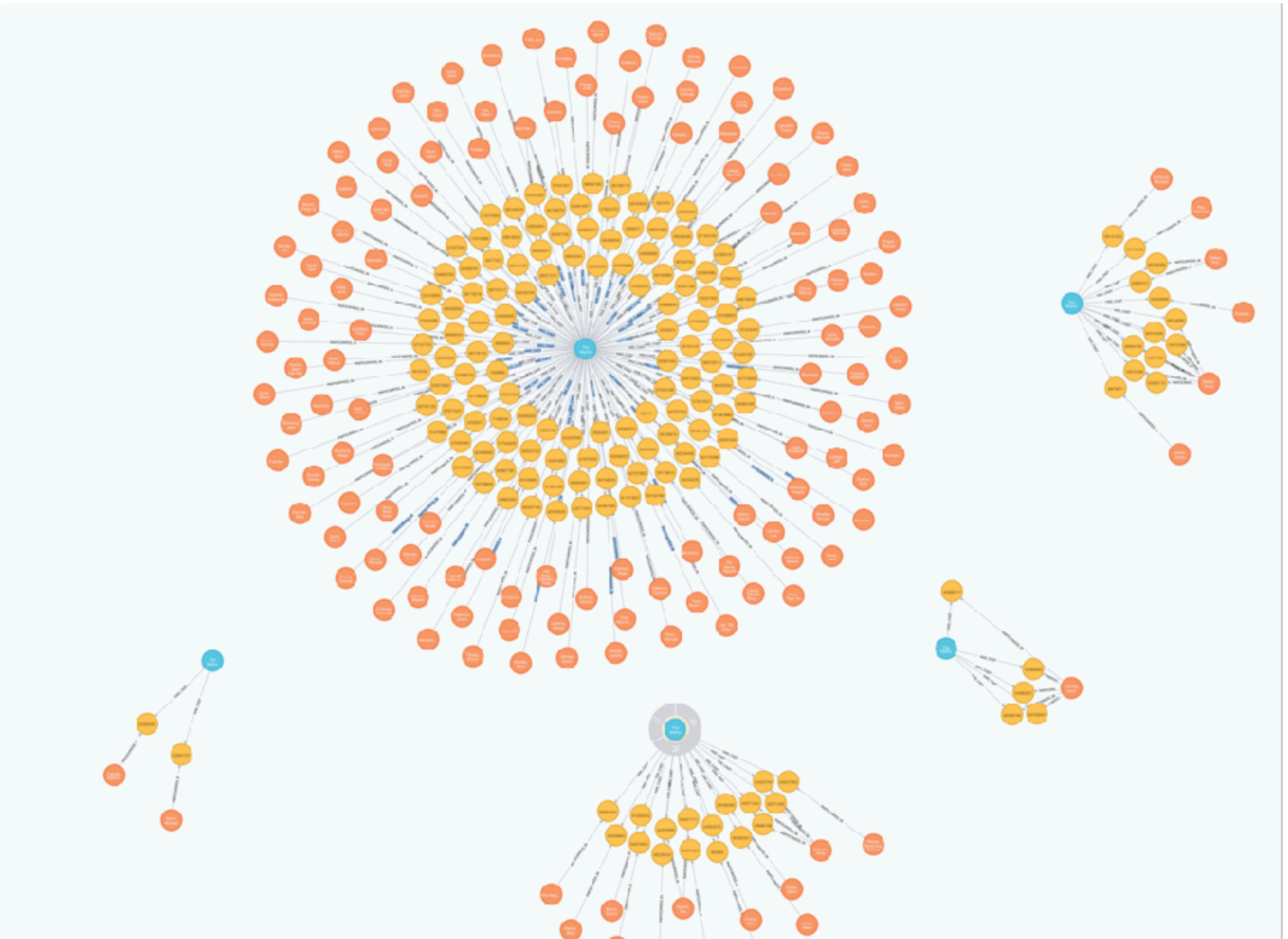
```
MATCH (t:Title{
           title: 'The Matrix',
           production_year: 1999}
      )-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
RETURN t,r,c,s,p
```



| t | r | c | s | p |
|---|---|---|---|---|
| {<br>"title_id": "4477062",<br>"phonetic_code": "M3623",<br>"kind_id": "1",<br>"title": "The Matrix",<br>"production_year": 1999<br>} | {<br>} | {<br>"cast_id": "63195892",<br>"movie_id": "4477062",<br>"role_id": "11",<br>"person_id": "1833362"<br>} | {<br>} | {<br>"name": "Paterson, Owen",<br>"name_pcode_cf": "P3625",<br>"gender": "m",<br>"name_pcode_nf": "O5136",<br>"imdb_index": "I",<br>"person_id": "1833362"<br>} |
| {<br>"title_id": "4477062",<br>"phonetic_code": "M3623",<br>"kind_id": "1",<br>"title": "The Matrix",<br>"production_year": 1999 | {<br>} | {<br>"note": "(kung fu choreographer) (as Yuen Wo Ping)",<br>"cast_id": "62758035",<br>"movie_id": "4477062",<br>"role_id": "10", | {<br>} | {<br>"name": "Yuen, Woo-Ping",<br>"surname_pcode": "Y5",<br>"name_pcode_cf": "Y5152",<br>"gender": "m",<br>"name_pcode_nf": "W1525", |

# EXAMPLES RETURN /5

```
MATCH (t:Title{
           title: 'The Matrix'}
      )-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
RETURN t,r,c,s,p
```



Returns multiple sub-graphs (potentially disconnected)

# WHERE CLAUSE /1

- Add constraints to the pattern to be matched
  - **Evaluation is at matching time**
- Usual operations
  - Checking on existing properties
  - Range ( <, <=, >, >= )
  - String matching (`starts with`, `ends with`, `contains`, regular expressions)
- Boolean Operations (`AND`, `OR`, `XOR`, `NOT`)
- Filters on node labels, node properties, relationship types, relationship properties

# EXAMPLES `WHERE` / CHECKING ON PROPERTIES, BOOLEAN OPERATIONS

```
MATCH (p:Person)
WHERE p.name_pcode_cf=‘A1652' AND p.name_pcode_nf='J2165'
RETURN p.name, p.gender
```

```
MATCH (p:Person{
         name_pcode_cf: ‘A1652',
         name_pcode_nf: ‘J2165’})
RETURN p.name, p.gender
```

✓

↕

✓

- Same semantics (same results set)

```
MATCH (p:Person)
WHERE p.name_pcode_cf=‘A1652'
                       AND (p.gender=‘m' OR p.gender=‘f')
RETURN p.name, p.gender
```

- No results with `null`

## EXAMPLES WHERE / STRING MATCHING WITH REGEX

```
MATCH (t:Title)
WHERE t.title=~ ’S.*’
RETURN t.title, t.production_year
```

- Titles having names starting with S

```
MATCH (t:Title)
WHERE t.title=~ '.r.*n'
RETURN t.title, t.production_year
```

- Titles with r as second letter and ending with n

```
MATCH (t:Title)
WHERE t.title=~ ‘.*Matrix.*’
RETURN t.title, t.production_year
```

- Titles containing the word Matrix

```
MATCH (t:Title)
WHERE t.name_pcode_cf=~ ‘.2*’
RETURN t.title, t.production_year
```

- Two characters, second is a 2

# WHERE CLAUSE /2

- Path patterns can also be predicates!
    - true if at least one solution is found
    - No new variables
- Path patterns in MATCH behave differently than path patterns in WHERE
    - MATCH: patterns produce subgraph for every found path
    - WHERE: it filters any matched sub-path where connected nodes have no relationship

```
MATCH (t:Title)
WHERE t.title=~ ‘.*Matrix.*’ AND (t)-[:HAS_CAST]-({note:'(creator)'})
RETURN t.title, t.production_year
```

## MISSING VALUES AND `NULL`

- Missing Values
  - A missing property evaluates to `null`
  - Checking for it in a node will make the condition evaluate to `false`

- `null` has the same semantics and properties as in SQL
  - `null` is not equal to `null`
  - `IS NULL` / `IS NOT NULL` work like in SQL

# EXAMPLES NULL

```
MATCH (p:Person)
WHERE p.imdb_index IS NULL
RETURN p.name, p.gender
```

```
MATCH (t:Title)
WHERE t.production_year > 2000 OR t.production_year IS NULL
RETURN t.title, t.production_year
```

# AGGREGATION FUNCTIONS

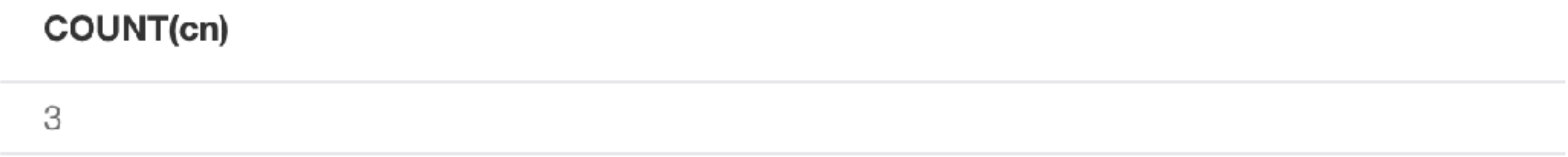
- Aggregation functions are analogous to SQL’s `GROUP BY`
  - Compute over all matching subgraphs
  - Non aggregate-expressions are used as group key
    - Matching sub-graphs are divided in buckets, according to the key value

```
RETURN t, count(*)
```

- Classic functions (`avg`, `count`, `max`, `min`, `sum`)
- `collect` returns a list containing the values returned by an expression

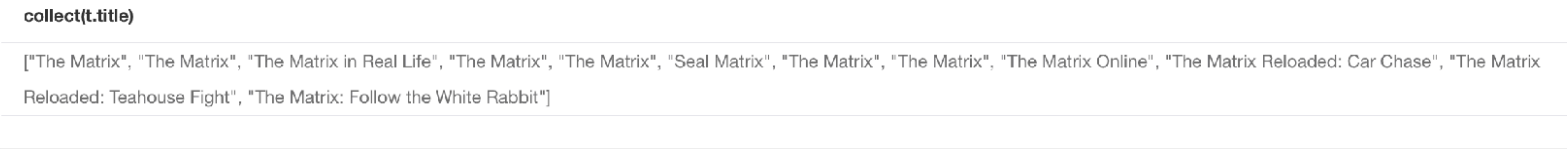
# EXAMPLE AGGREGATION FUNCTIONS

```
MATCH (cn:CharName{char_name:’Neo’})-[PERFORMED_AS]-(c:CastInfo)
RETURN COUNT(cn)
```



| COUNT(cn) |
|---|
| 3 |

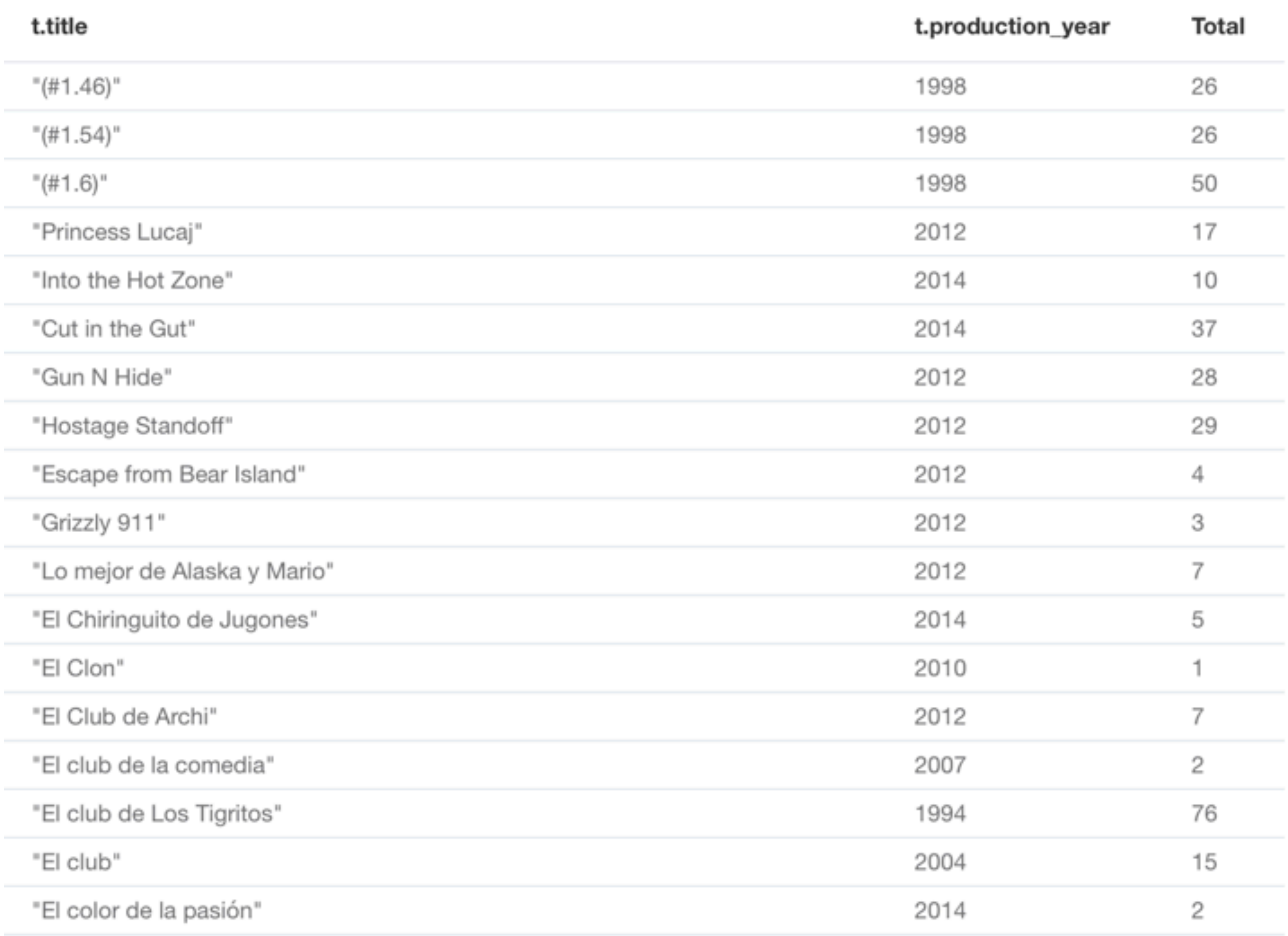
```
MATCH (t:Title)
WHERE t.title=~ ‘.*Matrix.*'
RETURN collect(t.title)
```



| collect(t.title) |
|---|
| ["The Matrix", "The Matrix", "The Matrix in Real Life", "The Matrix", "The Matrix", "Seal Matrix", "The Matrix", "The Matrix", "The Matrix Online", "The Matrix Reloaded: Car Chase", "The Matrix Reloaded: Teahouse Fight", "The Matrix: Follow the White Rabbit"] |

# EXAMPLE AGGREGATION FUNCTIONS / GROUP BY

```
MATCH (t:Title)-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
RETURN t.title, t.production_year, COUNT(c) AS Total
```



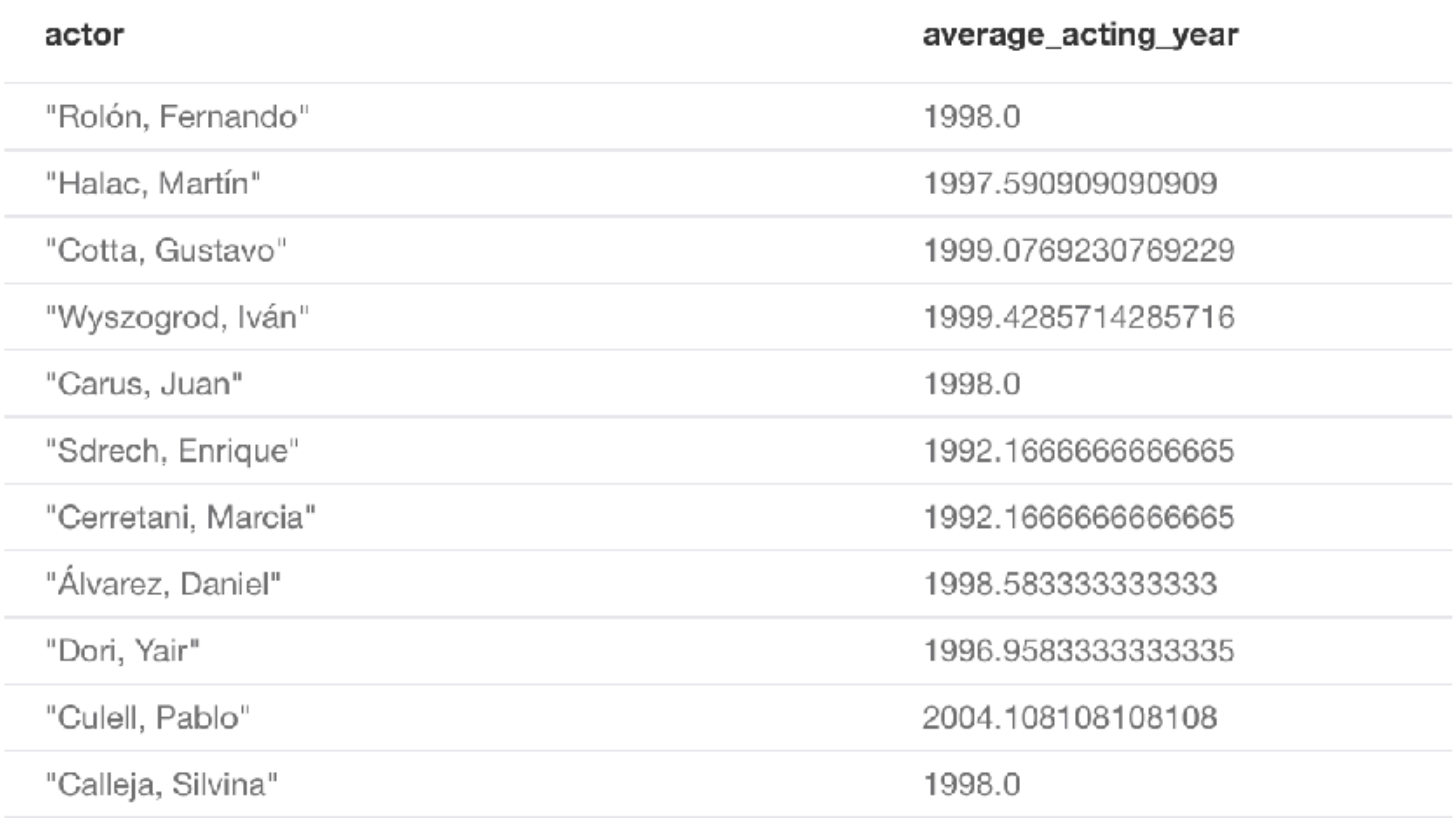
| t.title | t.production_year | Total |
|---|---|---|
| "(#1.46)" | 1998 | 26 |
| "(#1.54)" | 1998 | 26 |
| "(#1.6)" | 1998 | 50 |
| "Princess Lucaj" | 2012 | 17 |
| "Into the Hot Zone" | 2014 | 10 |
| "Cut in the Gut" | 2014 | 37 |
| "Gun N Hide" | 2012 | 28 |
| "Hostage Standoff" | 2012 | 29 |
| "Escape from Bear Island" | 2012 | 4 |
| "Grizzly 911" | 2012 | 3 |
| "Lo mejor de Alaska y Mario" | 2012 | 7 |
| "El Chiringuito de Jugones" | 2014 | 5 |
| "El Clon" | 2010 | 1 |
| "El Club de Archi" | 2012 | 7 |
| "El club de la comedia" | 2007 | 2 |
| "El club de Los Tigritos" | 1994 | 76 |
| "El club" | 2004 | 15 |
| "El color de la pasión" | 2014 | 2 |

## THE WITH CLAUSE

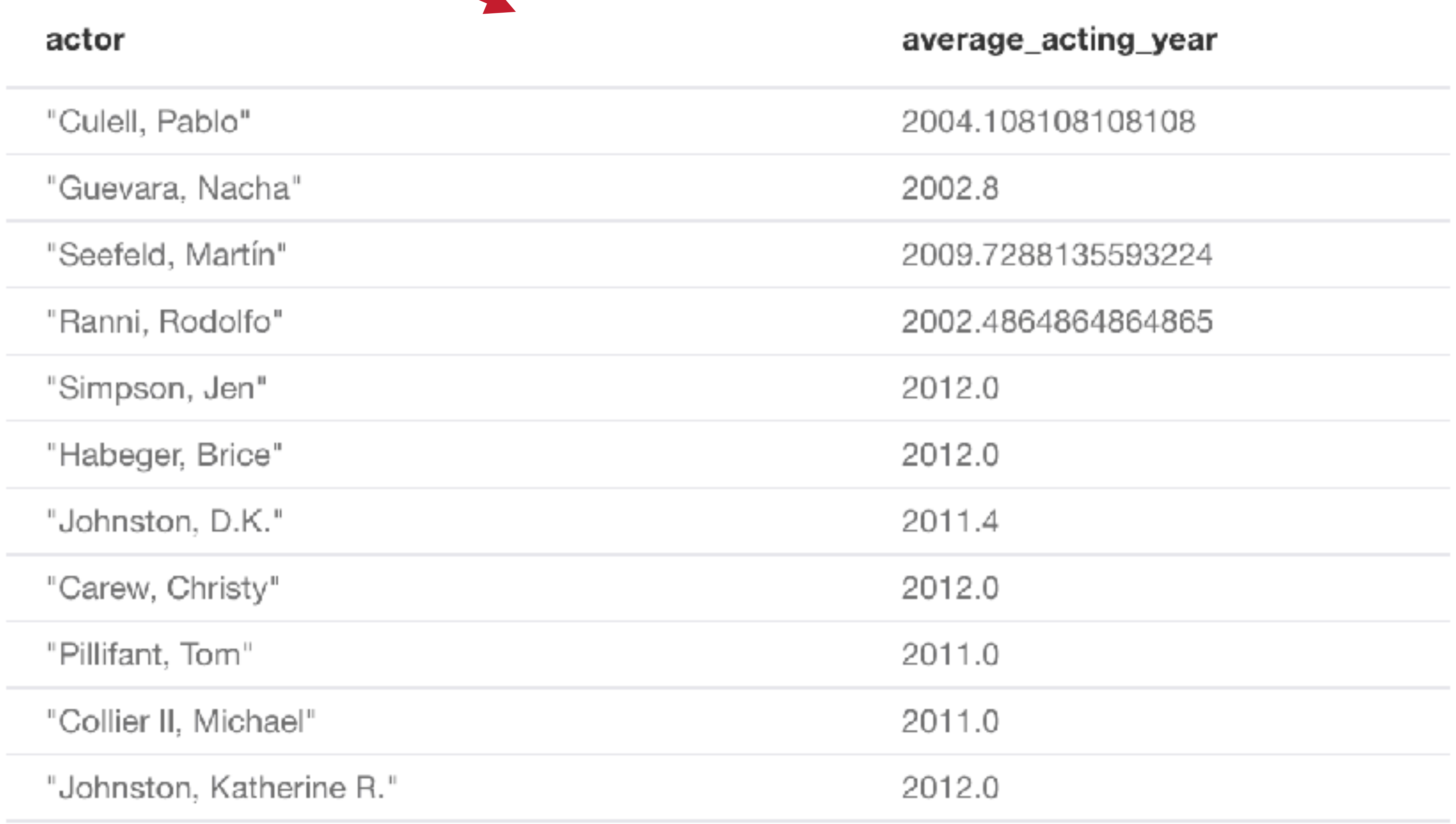
- Behaves analogously to the RETURN clause
- Does not output anything to the user, just forwards the current result to the subsequent clause
    - The output of one part is passed as input to another
- Useful to filter on aggregate values (there is no HAVING clause in Cypher)
- But it can also be used to decompose queries into “sub-queries”

# EXAMPLE WITH CLAUSE / HAVING

```
MATCH (t:Title)-[r:HAS_CAST]->(c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
WITH p.name AS actor, avg(t.production_year) AS average_acting_year
WHERE average_acting_year > 2000
RETURN actor, average_acting_year
```



| actor | average_acting_year |
|---|---|
| "Rolón, Fernando" | 1998.0 |
| "Halac, Martín" | 1997.590909090909 |
| "Cotta, Gustavo" | 1999.0769230769229 |
| "Wyszogrod, Iván" | 1999.4285714285716 |
| "Carus, Juan" | 1998.0 |
| "Sdrech, Enrique" | 1992.1666666666665 |
| "Cerretani, Marcia" | 1992.1666666666665 |
| "Álvarez, Daniel" | 1998.583333333333 |
| "Dori, Yair" | 1996.9583333333335 |
| "Culell, Pablo" | 2004.108108108108 |
| "Calleja, Silvina" | 1998.0 |



| actor | average_acting_year |
|---|---|
| "Culell, Pablo" | 2004.108108108108 |
| "Guevara, Nacha" | 2002.8 |
| "Seefeld, Martín" | 2009.7288135593224 |
| "Ranni, Rodolfo" | 2002.4864864864865 |
| "Simpson, Jen" | 2012.0 |
| "Habeger, Brice" | 2012.0 |
| "Johnston, D.K." | 2011.4 |
| "Carew, Christy" | 2012.0 |
| "Pillifant, Tom" | 2011.0 |
| "Collier II, Michael" | 2011.0 |
| "Johnston, Katherine R." | 2012.0 |

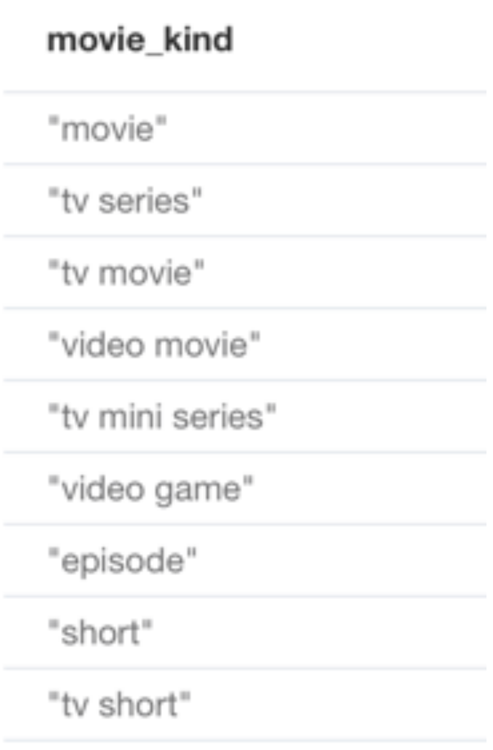
# EXAMPLE WITH CLAUSE / “SUB QUERIES”

✓

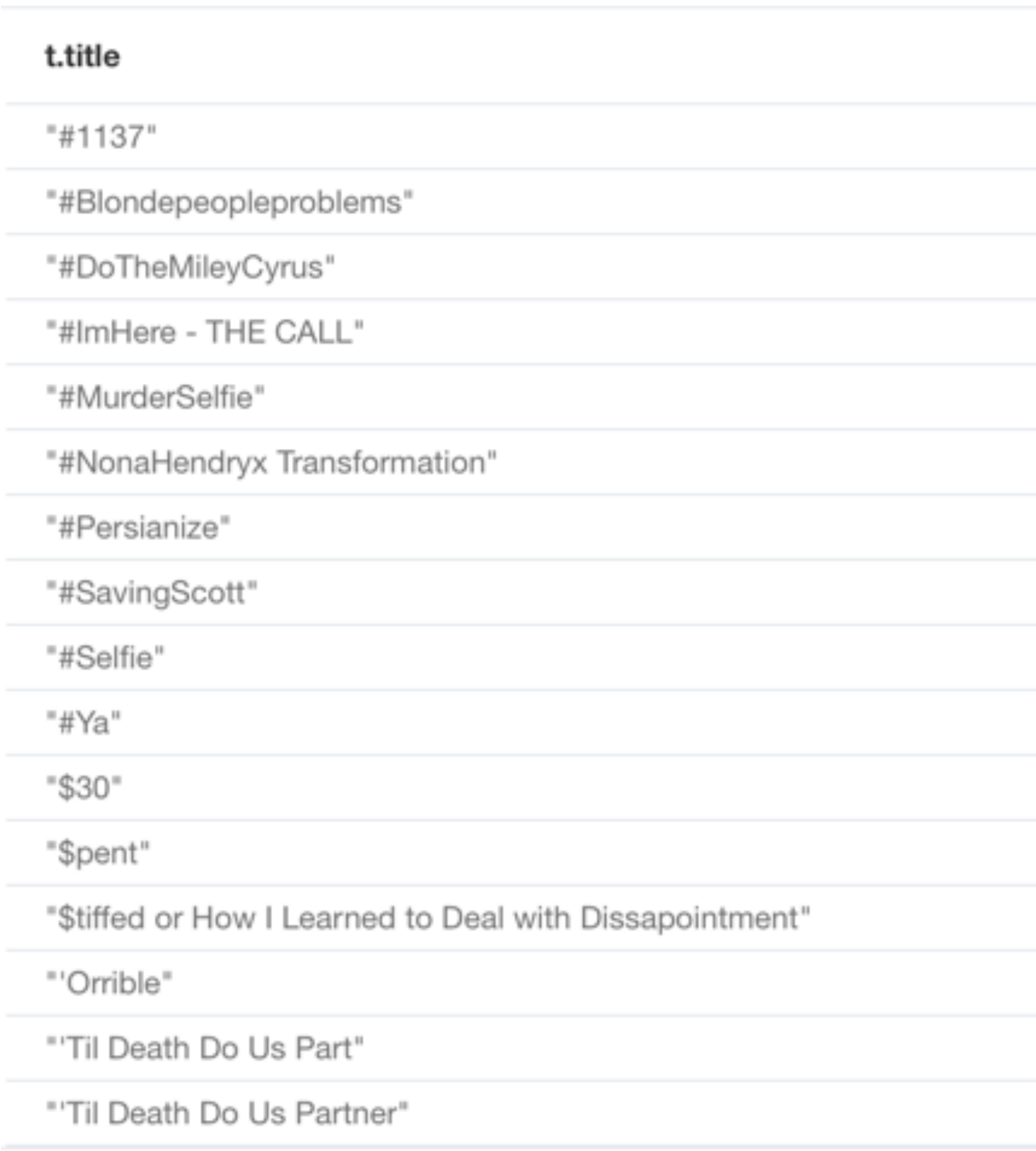
```
MATCH (t:Title)-[i:IS_OF_KIND]->(k:KindType{kind_name:’movie’})
RETURN t.title
```

✓

```
MATCH (k:KindType{kind_name:’movie’})
WITH k.kind_id AS movie_kind
MATCH (t:Title)
WHERE t.kind_id = movie_kind
RETURN t.title
```



| movie_kind |
|---|
| "movie" |
| "tv series" |
| "tv movie" |
| "video movie" |
| "tv mini series" |
| "video game" |
| "episode" |
| "short" |
| "tv short" |



| t.title |
|---|
| "#1137" |
| "#Blondepeopleproblems" |
| "#DoTheMileyCyrus" |
| "#ImHere - THE CALL" |
| "#MurderSelfie" |
| "#NonaHendryx Transformation" |
| "#Persianize" |
| "#SavingScott" |
| "#Selfie" |
| "#Ya" |
| "$30" |
| "$pent" |
| "$tiffed or How I Learned to Deal with Dissapointment" |
| "'Orrible" |
| "'Til Death Do Us Part" |
| "'Til Death Do Us Partner" |

# UNION

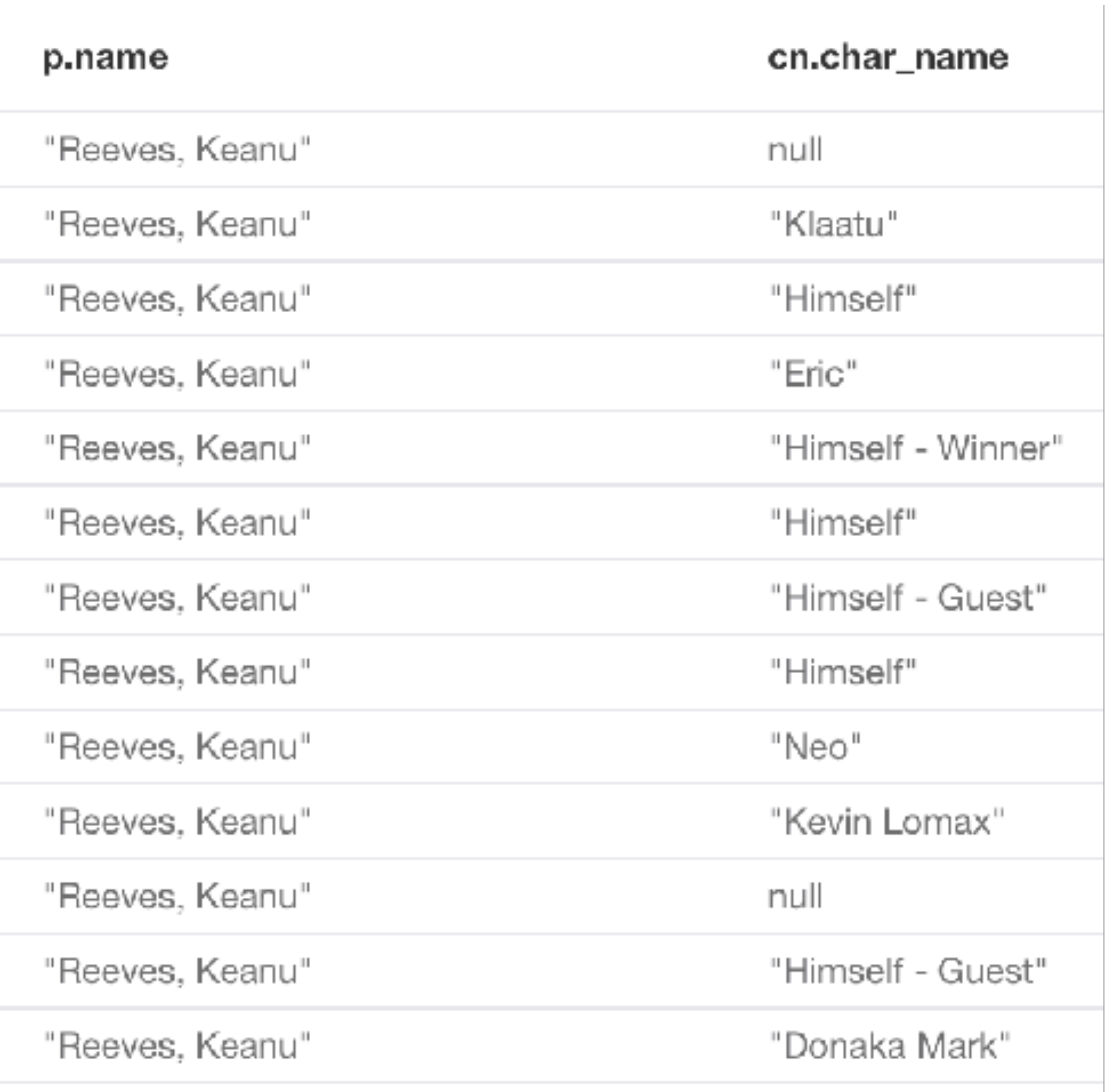
- Combines the result of multiple queries
- Number and names of columns must be identical in all queries
- `UNION ALL` keeps duplicates

```
  MATCH (p:Person)
  RETURN p.name
UNION
  MATCH (t:Title)
  RETURN t.title
```

# OPTIONAL MATCH

- When no solution is found, one specific solution with all the variables bound to NULL is generated
  - Pretty much a LEFT join
  - Either the whole pattern is matched, or nothing is matched

```
MATCH (c:CastInfo)<-[s:PARTICIPATES_IN]-(p:Person)
WHERE p.name = ‘Reeves, Keanu’
OPTIONAL MATCH (c)-[pa:PERFORMED_AS]->(cn:CharName)
RETURN p.name, cn.char_name
```



| p.name | cn.char_name |
|---|---|
| "Reeves, Keanu" | null |
| "Reeves, Keanu" | "Klaatu" |
| "Reeves, Keanu" | "Himself" |
| "Reeves, Keanu" | "Eric" |
| "Reeves, Keanu" | "Himself - Winner" |
| "Reeves, Keanu" | "Himself" |
| "Reeves, Keanu" | "Himself - Guest" |
| "Reeves, Keanu" | "Himself" |
| "Reeves, Keanu" | "Neo" |
| "Reeves, Keanu" | "Kevin Lomax" |
| "Reeves, Keanu" | null |
| "Reeves, Keanu" | "Himself - Guest" |
| "Reeves, Keanu" | "Donaka Mark" |

# ORDER BY

- By default, order of results is not defined
- Multiple criteria can be specified
  - Default direction is ASC
- Sorting is allowed only on properties
  - Not on nodes and relationships

```
MATCH (t:Title)
RETURN t
ORDER BY t.production_year DESC, t.title ASC
```

# WAYS TO IMPROVE PERFORMANCE

- Use parameters instead of literals when possible
  - Cypher can re-use the queries
- Set upper limit for variable length pattern
- Return only the data you need
  - Avoid returning whole nodes and relationships
- Use PROFILE / EXPLAIN to analyse the performance of your queries
- Know/learn the types of questions to answer, then create new relationships that speed up those questions
- Use indexes

# OTHER THINGS YOU CAN DO

- Depth and Breadth-First Search
- Path Finding (e.g. Dijkstra, A*)
- Triadic Closure
- In general, network properties (e.g. centrality, betweenness, etc.)

# WRAPPING UP

## TODAY WE COVERED

- Neo4J: data model and query language

# END OF LECTURE