Neo4j Loaded

A look into graph databases
with
Neo4j

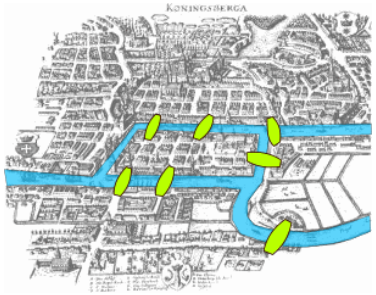
Agenda

- Introduction to graph databases
- Graph VS Relational databases
- Modeling concepts
- Neo4j installation / tools
- Introduction to Cypher
- :play movies

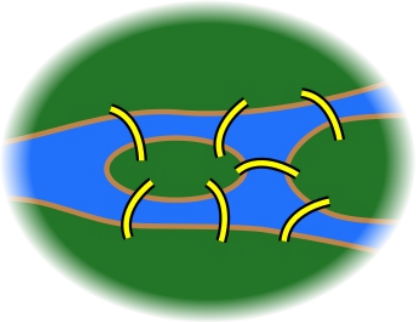
History

- Graph theory may be first traced in 1736
- Leonard Euler (Swiss) solved the Königsberg bridge problem:

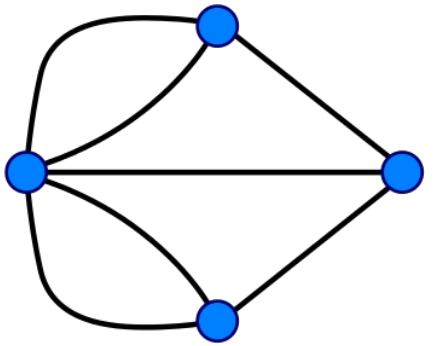
> Find a walk through the city that will cross each bridge exactly once.



Map of Kaliningrad, Russia



- Moves inside an island are irrelevant
- Route is a sequence of bridges
- Abstract over land masses and bridge connections



- Nodes (Vertices)
- Edges (Relationships)

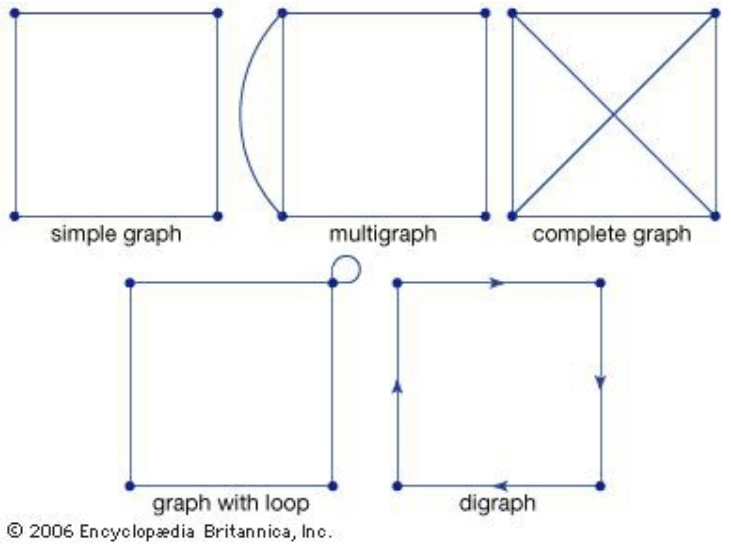
Source: Wikipedia

What is a Graph ?

A Graph is a collection of vertices/nodes and edges/relationships.

Graphs represent entities as **nodes** and the ways in which they are related as **relationships**

- ✖ Data charts are NOT graphs
- ✖ Cannot have relationships w/o adjacent nodes
- ✔ Can have nodes w/o relationships



What is a graph database ?

A database where:

- **nodes**, **edges** and **properties** are used to represent and store data
- queries are defined by graph structures

Similarly to graphs:

- **nodes** are used to represent entities
- **edges** are used to represent relationships between nodes
- **properties** contain information related to nodes and/or relationships

Relationships in graph databases are first class citizens

The word relational in RDBMS stems from relational algebra and not from relationship.

Graph Vs Relational databases (1/3)

- Relational databases are a good fit for:
  - data that are predictable (follow a certain schema)
  - operating on a huge number of records
  - no join-intensive queries (multiple joins, self-joins or recursive/hierarchical ones)
  - applying business rules in database level
  - providing survivor functionality / data quality capabilities

<u>But:</u>

- Data become more and more interconnected
- Data structures change
- RealTime response times are crucial

Graph Vs Relational databases (2/3)

=> Graph DB's: Next generation of relational databases

- Relationships are first-class citizens
  - Pre-materialized as connections between nodes
- Relationships are more expressive
  - Types, directions on links, attributes
- Performant in identifying patterns/relationship **on a single individual**
  - No need for nested joins
- Whiteboard-friendly modeling
  - Logical model is most likely the physical model
- Flexible – easy to extend
  - lack of normalization tables

Graph Vs Relational databases (3/3)

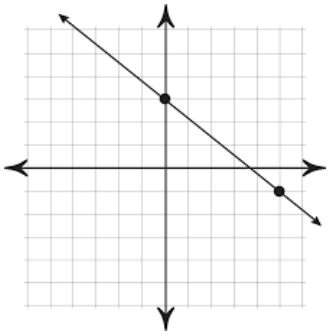
	RDBMS	Graph DB's
Storage capabilities	★★★	★★★
Analytic queries / queries on massive data	★★	★
Pattern identification / complex queries on individuals	★	★★★
Transaction time	★	★★
Modeling / Expressiveness	★	★★★
Data integrity / quality mechanism	★★★	★
Flexibility / extensibility	★	★★
Write queries	★	★★

Quiz

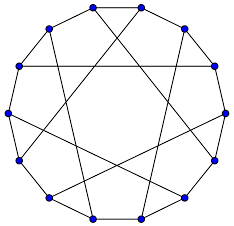
- Which one of the below is a graph ?

A)

B)



C)



Quiz

- Relationships are first class citizens for

  a) relational databases

  b) graph databases

Quiz

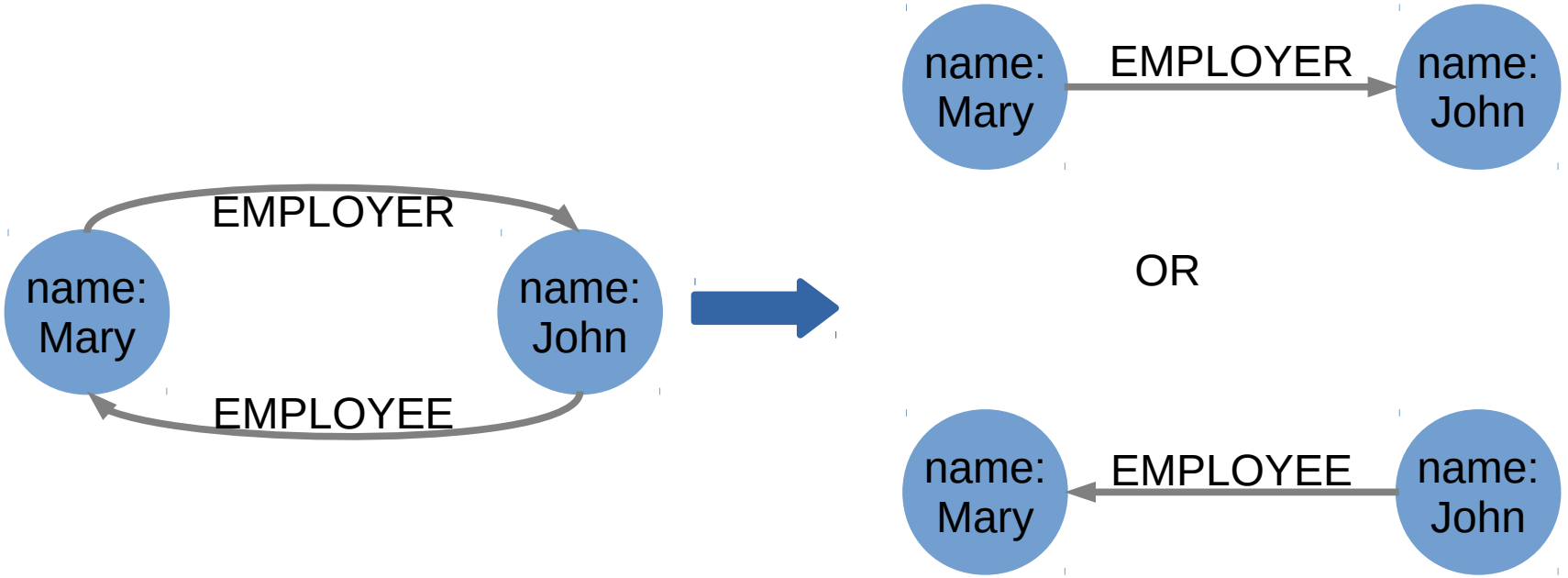
- Graph databases outperform relational databases when:
  1) data are predictable
  2) operating on a huge number of records
  3) having join-intensive queries
  4) need a flexible db schema
  5) providing survivor functionality / data quality capabilities

The Neo4j graph database

- Labeled property graph model
  - Nodes can have **attributes** (key/value pairs) and **labels**
    - labels can be used to represent roles of the domain
    - labels can be used to define indexes/constraint info
  - Relationships have **directions**
  - Relationships can have **properties** and **labels** too
  - Relationships always connect **two** nodes
- ACID transactions
- Constant time traversals and graph caching
- Written on top of JVM (Java mainly and Scala)
- Free open-source Community edition
- Enterprise edition with scalable clustering, fail-over, high-availability, live backups, and comprehensive monitoring features

Graph Data Modeling Guidelines

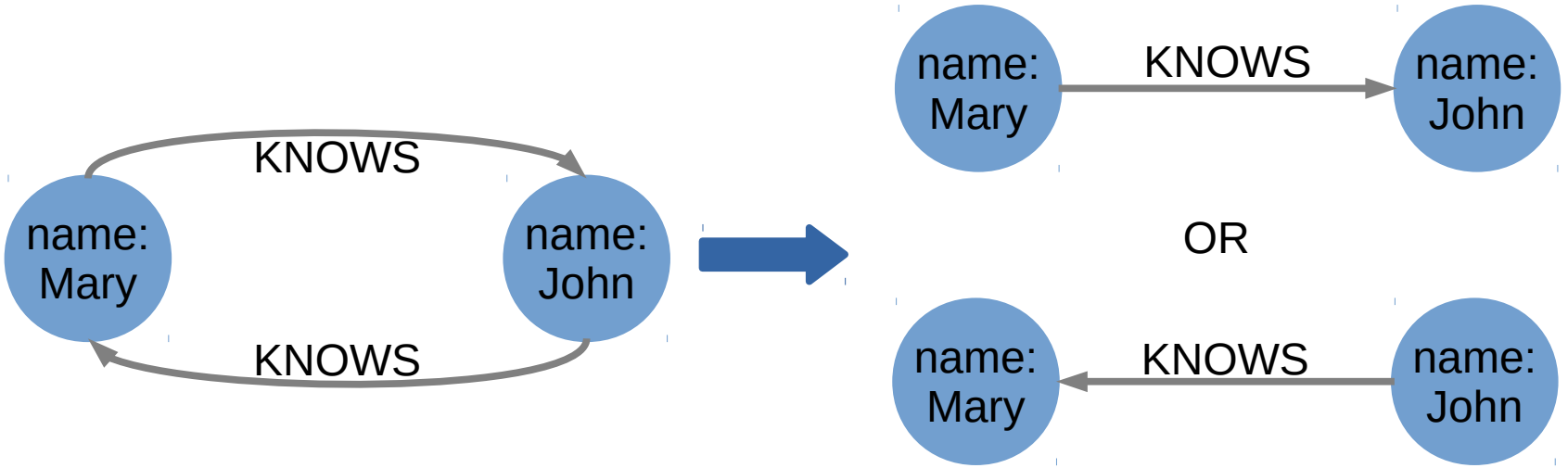
- Symmetric relationships



In case one relationship is implied by the presence of another one then we can leave one of them to be inferred.

Graph Data Modeling Guidelines

- Bi-Directional relationships
  - Neo4j relationships have single direction



In case Mary knowing John implies that John also knows Mary then one relationship in any direction is enough. In queries we can ignore relationship direction.

Graph Data Modeling Guidelines

- Properties VS Relationships

  2 extreme approaches:

  - **Fine-grained** graph (exploit **relationship** elements)
  - **Coarse-grained** graph (exploit **property** elements)

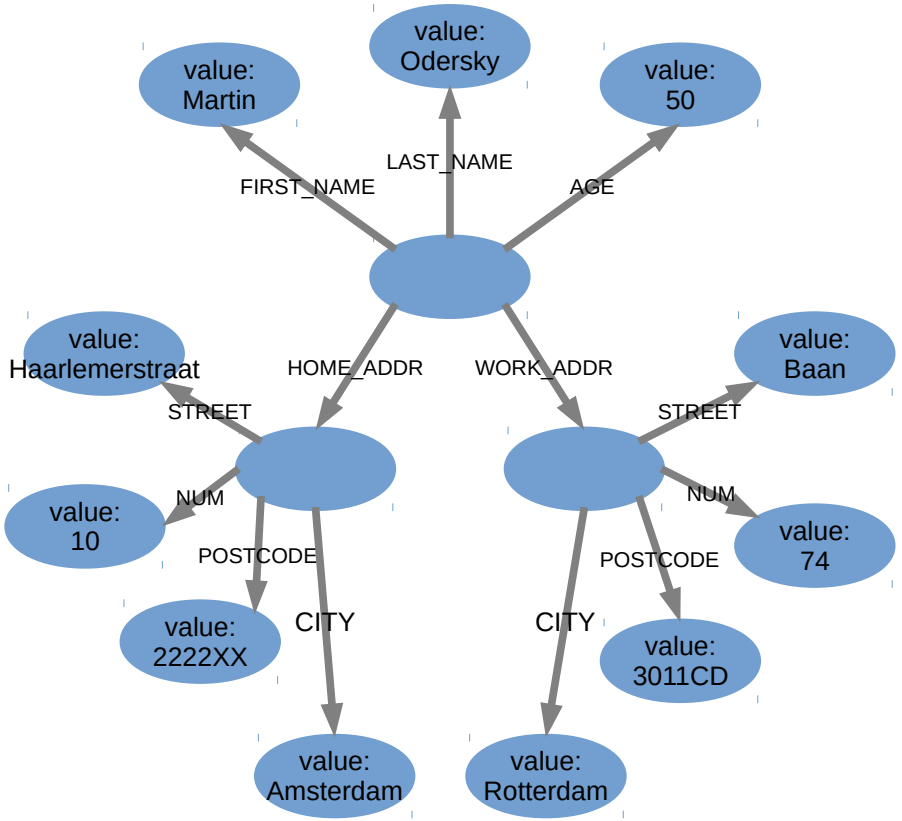
  Exercise: Model the record of Lunatech employees following each approach

  - first name
  - last name
  - age
  - home address (street, number of street, city, post code)
  - office address (street, number of street, city, post code)

Graph Data Modeling Guidelines

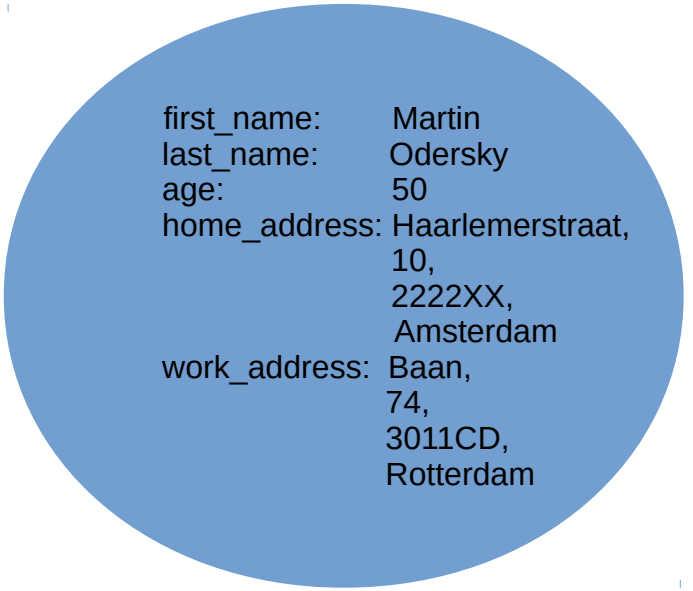
- Properties VS Relationships

  Exercise: Model the record of Lunatech employees following each approach



Fine-grained graph

Coarse-grained graph



Graph Data Modeling Guidelines

- In general use relationships when :
  - need to specify weight, strength or any other quality of a relationship
    - Proficiency in a skill
    - Friendship strength
  - attribute values create a complex type
    - Address(street, number, post code, city)
  - attribute values are interconnected, i.e. other nodes might need to connect to it
    - A taxonomy of skills

No best practices in modeling a graph

Graph Data Modeling Guidelines

Steps:

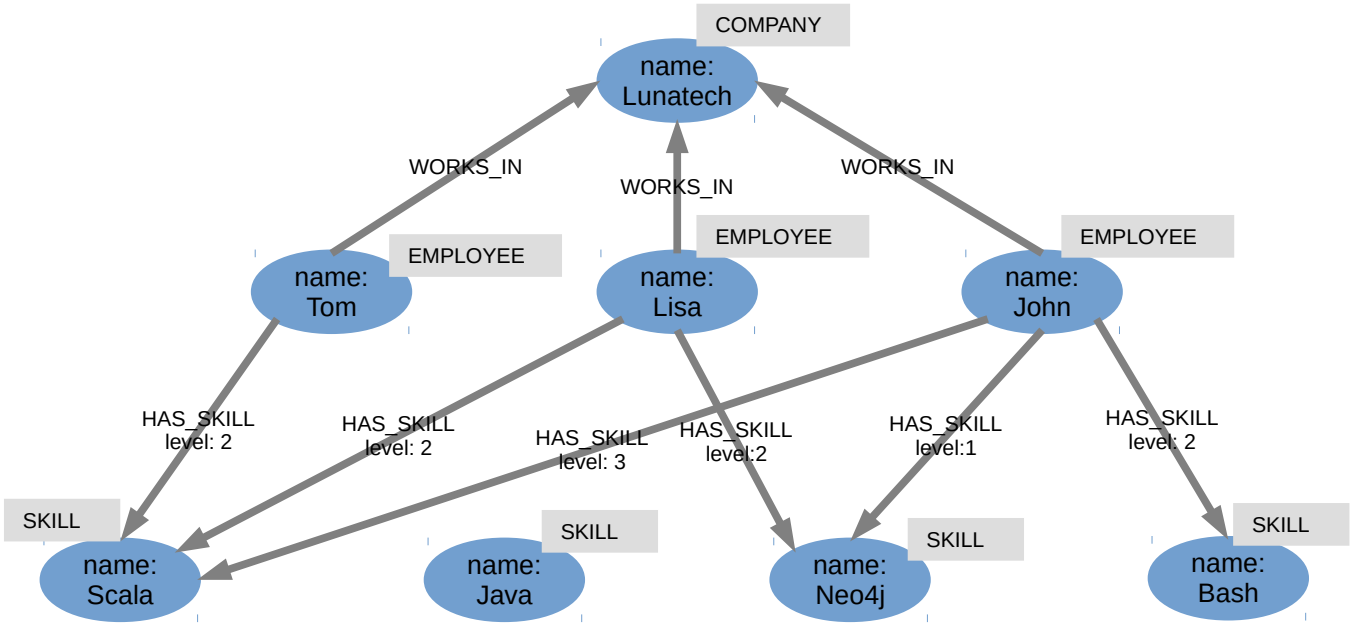
1) Describe your domain.
2) Identify the questions that your model needs to answer.
3) Define your entities and relationships between them.
4) Place the attributes in your model that are necessary for answering your questions.
5) Address questions like:
   - Can this attribute value exist independently of the owning node?
   - Will this attribute value comprise a starting point for a query?
   - Do you want to relate to the attribute?
6) Create a sample dataset.
7) Translate questions into queries.
8) Profile your queries (could also be done later on a much bigger dataset) .
9) Repeat 3 – 8 if necessary.

Graph Data Modeling Guidelines

Exercise:

Model the skills that each employee has working in a company

Q1: Find the level of expertise an employee has on his skills

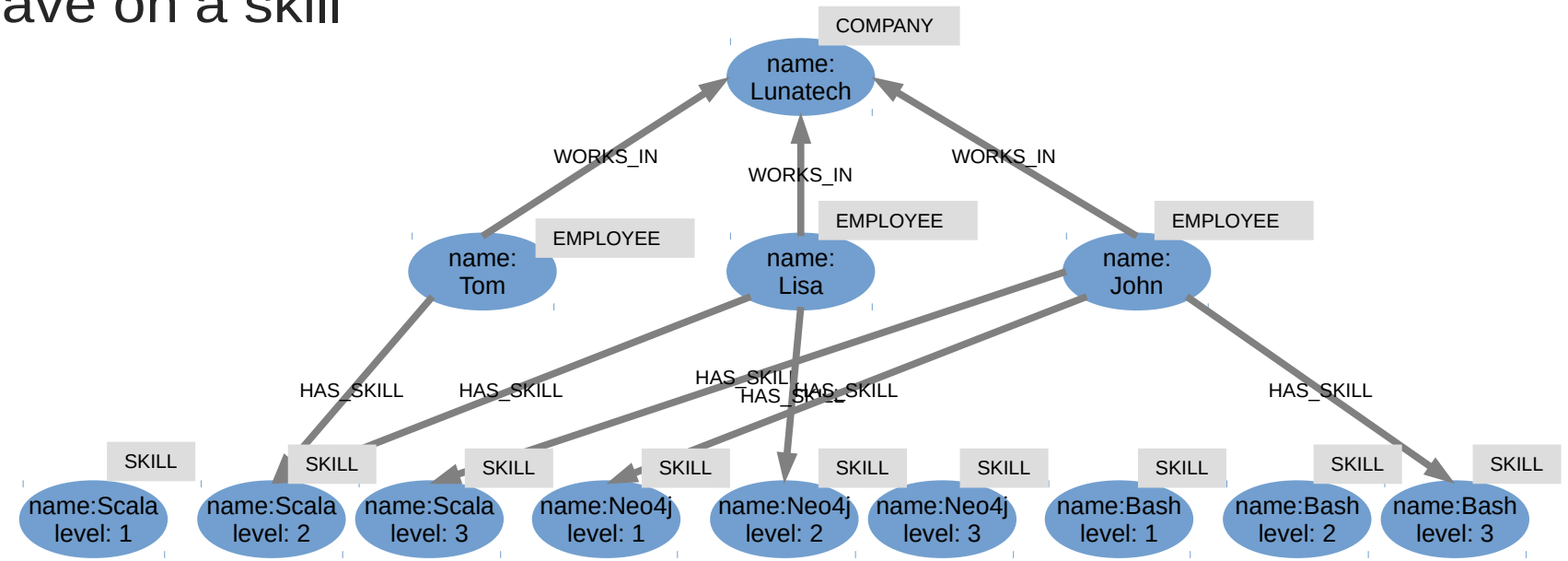


Graph Data Modeling Guidelines

Exercise:

Model the skills that each employee has working in a company

Q2: Find the employees depending on the level of expertise they have on a skill



Install Neo4j

A movies graph database

1) **Describe your domain.**

We want to have a database that stores information about movies and the persons that contributed it.
A person may have written, produced, directed or acted in a movie.

2) **Identify the questions that your model needs to answer (optimally).**

- Which movies (title, summary, duration) belong to a specific genre ?

- Which are the movies (title, tagline, rating) having a specific keyword ?

  → a movie can correspond to multiple genres

  → a movie can have multiple keywords related to it

  → genres and keywords may correspond to multiple movies

- What is the date of birth of an person (search on his/her name) ?

- What is the role a person that acted in a movie played ?

- How many roles did a person play in a movie ?

- Who are the contributors of a movie ?

A movies graph database

1) Describe your domain.

We want to have a database that stores information about **Movies** and the **Persons** that contributed to it.
A **Person** may have written, produced, directed or acted in a **Movie**.

2) Identify the questions that your model needs to answer (optimally).

- Which **Movies** (*title*, *summary*, *duration*) belong to a specific **Genre** ?

- Which are the **Movies** (*title*, *tagline*, *rating*) having a specific **Keyword** ?

  → a **Movie** can have multiple Genres

  → a **Movie** can have multiple Keywords related to it

  → **Genres** and **Keywords** may correspond to multiple **Movies**

- What is the *date of birth* of an **Person** (search on his/her *name*) ?

- What is the *role* a **Person** that acted in a **Movie** played ?

- How many roles did a **Person** play in a **Movie** ?

- Who are the **Persons** that contributed in a **Movie** ?

A movies graph database

3) Define your entities and relationships between them.

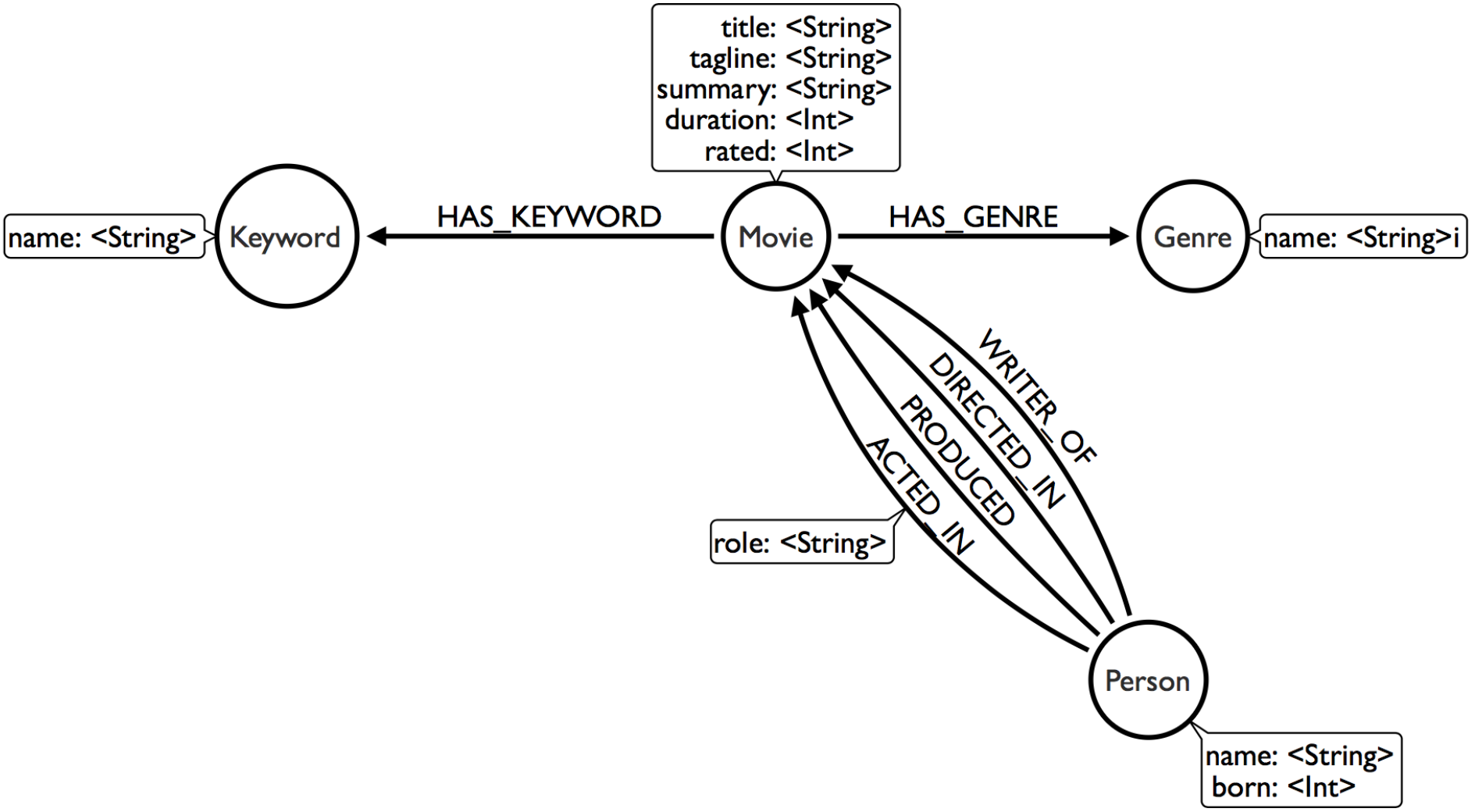
4) Place the attributes in your model that are necessary for answering your questions.

5)Address questions like:

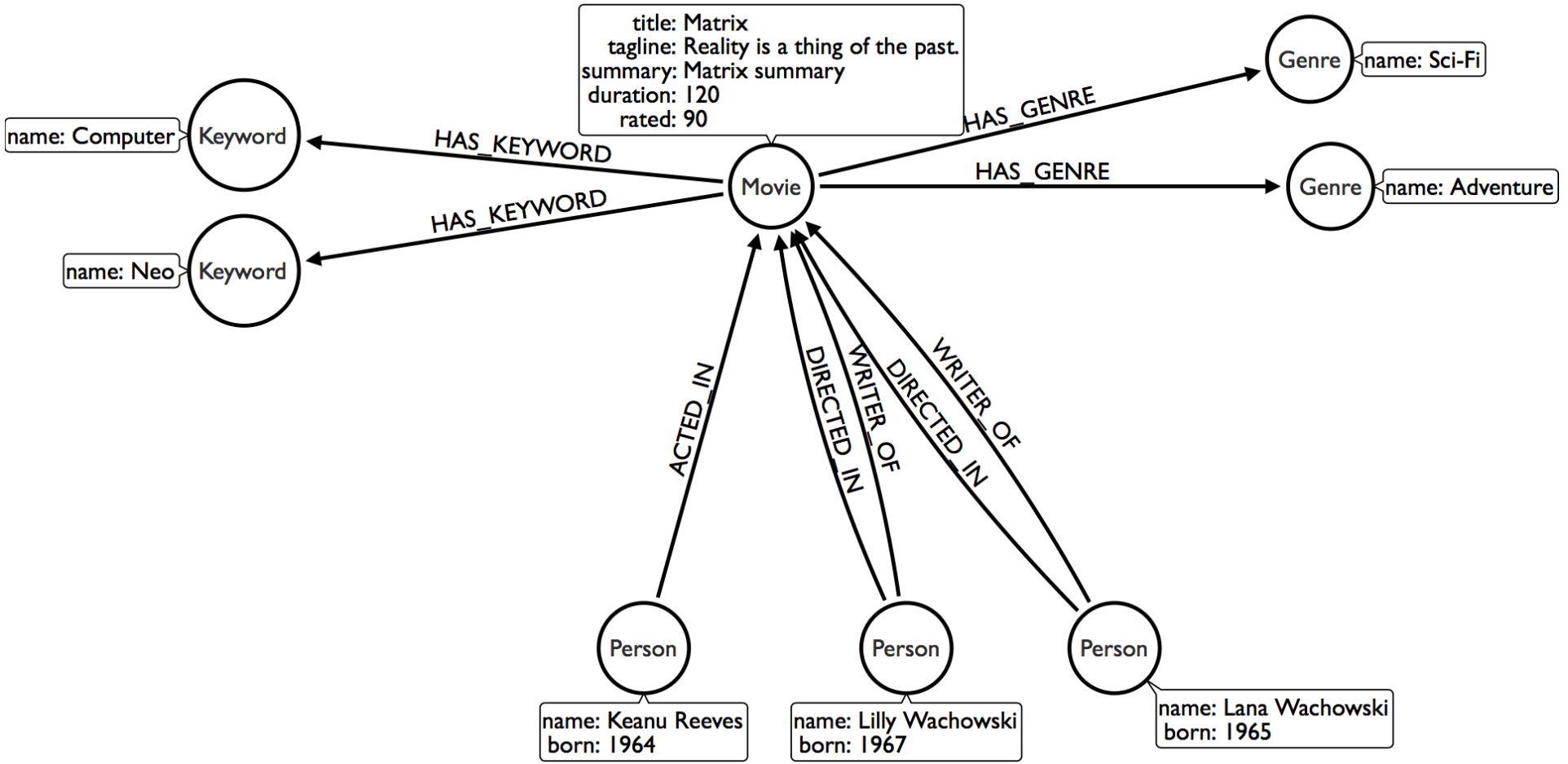
- Can this attribute value exist independently of the owning node?
- Will this attribute value comprise a starting point for a query?
- Do you want to relate to the attribute?

6) Create a sample dataset

A movies graph database



A movies graph database



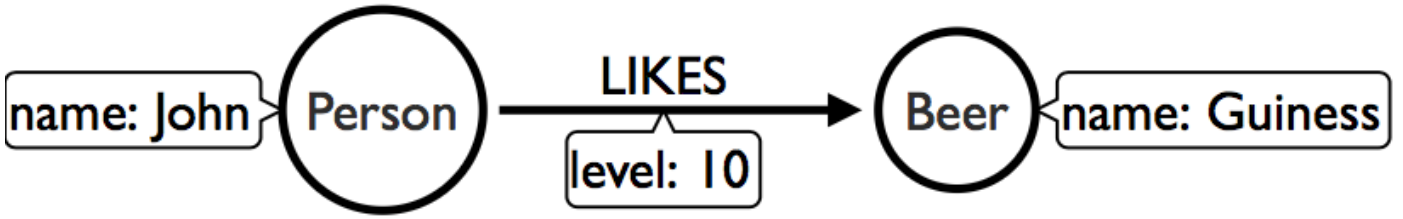
Introduction to Cypher

Cypher is a declarative, SQL-inspired language for describing patterns in graphs visually using an ascii-art syntax.

It allows us to state what we want to select, insert, update or delete from our graph data without requiring us to describe exactly how to do it

Neo4j.com

Querying with cypher is all about defining the right pattern:

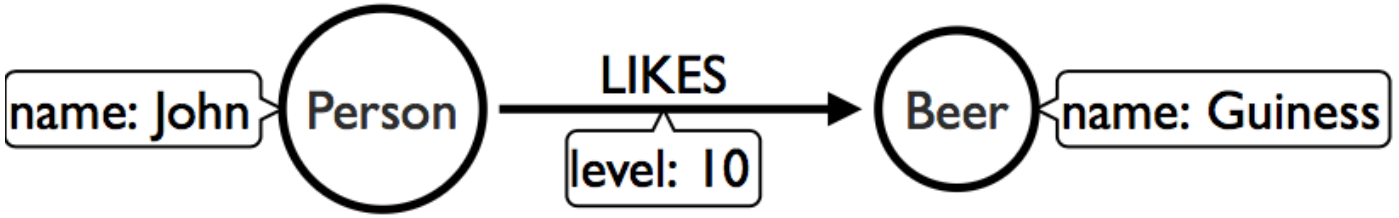



```
(anyone)                          -->                  (anything)
(anyone)                          -[:LIKES]->          (anything)
(person: Person)                  -[:LIKES]->          (beer: Beer)
(john: Person {name: 'John'})     -[:LIKES]->          (beer: Beer)
(john: Person {name: 'John'})     -[:LIKES]->          (guiness: Beer {name: 'Guiness'})
(john: Person {name: 'John'})     -[:LIKES {level: 10} ]-> (guiness: Beer {name: 'Guiness'})
```

- **All these patterns match the above graph structure but return different results in total**

Introduction to Cypher

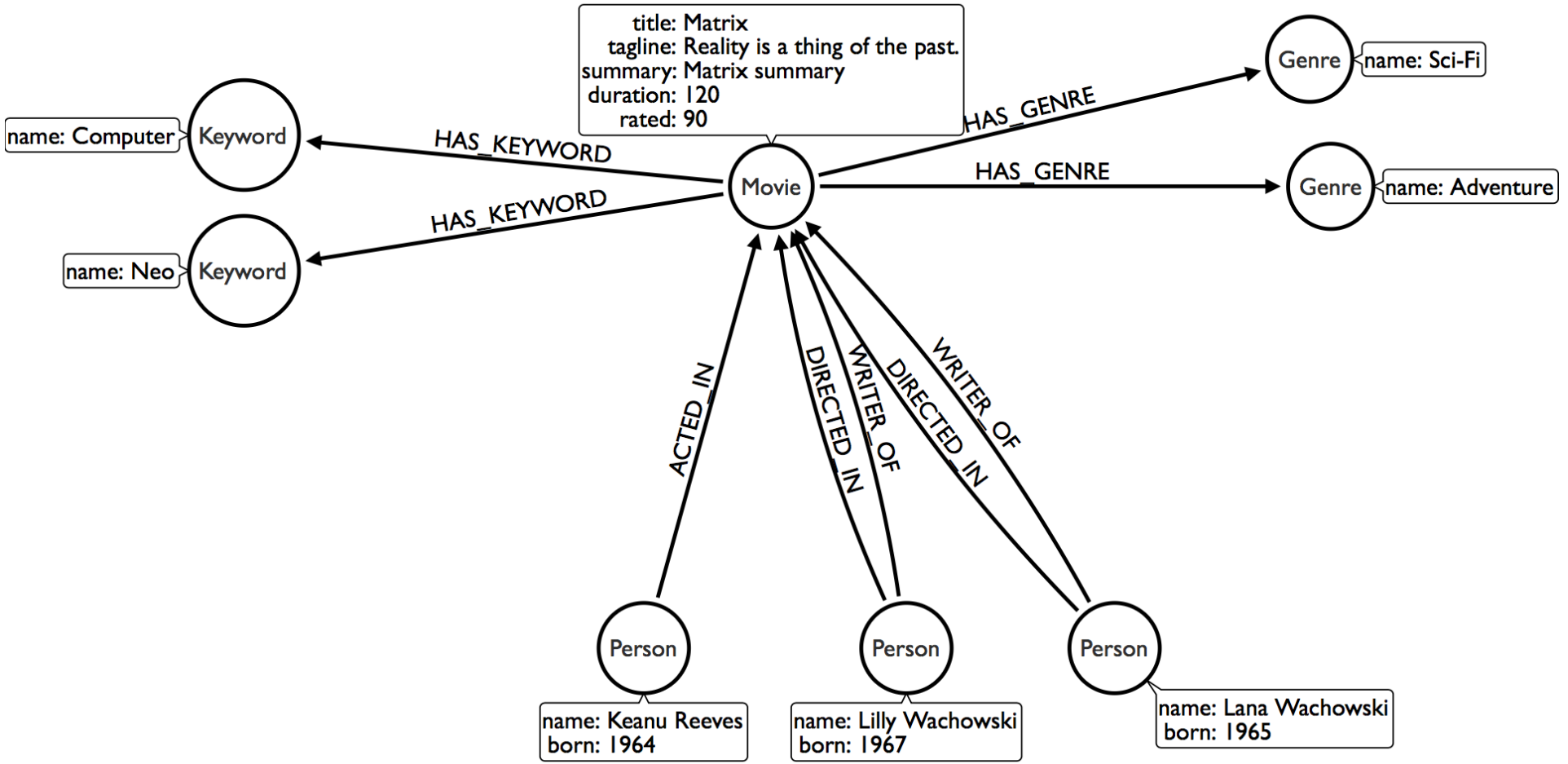
- Create nodes and relationships with Cypher




```
CREATE
(john: Person {name: 'John'}) ,
(guiness: Beer {name: 'Guiness'}) ,
(john)-[:LIKES {level: 10}]->(guiness)
```

→ We assign the new nodes in variables (john, guiness) and then we create the relationship between them

Exercise: Create a sample movies database



Exercise: Create a sample movies database

```
CREATE
(keanu: Person {name: "Keanu Reeves", born: 1964}),
(lana: Person {name: "Lana Wachowski", born: 1965}),
(lilly: Person {name: "Lilly Wachowski", born: 1967}),
(matrix: Movie {title: "Matrix", tagline: "Reaity is a thing of the past"}),
(computer: Keyword {name: "Computer"}),
(neo: Keyword {name: "Neo"}),
(scifi: Genre {name: "Sci-Fi"}),
(adv: Genre {name: "Adventure"}),
(keanu)-[:ACTED_IN]->(matrix),
(lana)-[:WRITER_OF]->(matrix),
(lana)-[:DIRECTED_IN]->(matrix),
(lilly)-[:WRITER_OF]->(matrix),
(lilly)-[:DIRECTED_IN]->(matrix),
(computer)<-[:HAS_KEYWORD]-(matrix),
(neo)<-[:HAS_KEYWORD]-(matrix),
(matrix)-[:HAS_GENRE]->(scifi),
(matrix)-[:HAS_GENRE]->(adv)
```

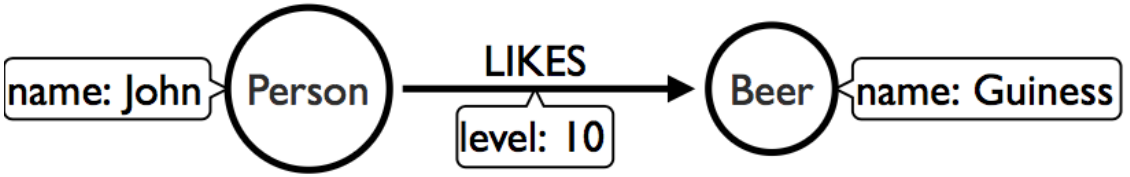
Use the neo4j-import tool

- Load movies database

https://github.com/dcharoulis/neo4j-workshop

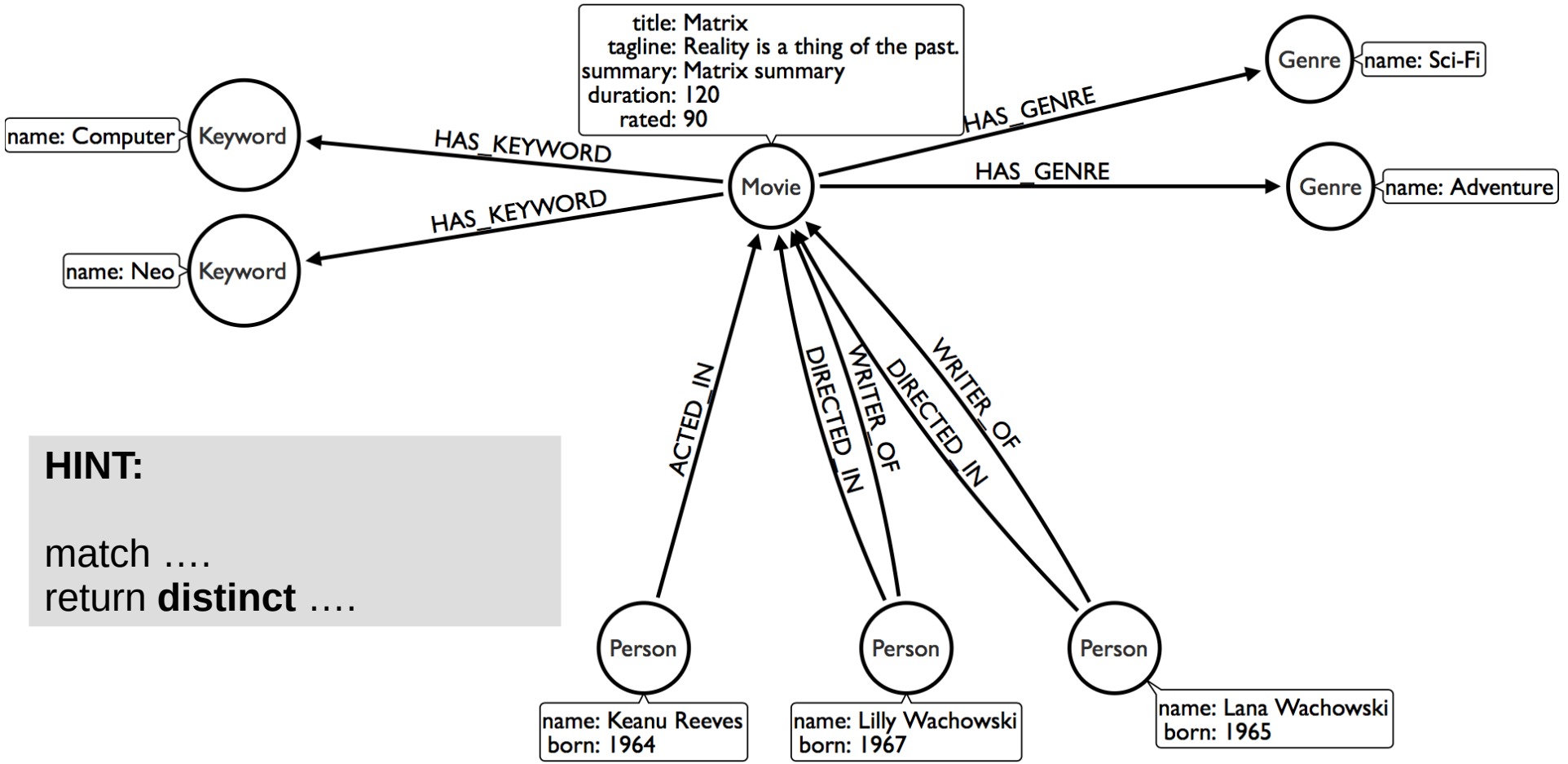
Introduction to Cypher

- Matching patterns on a graph




```
// match any two related nodes, direction not significant
MATCH (a)--(b)
RETURN a,b

//Same as above, we are only interested in start node of the relationship
MATCH (a)-->()
RETURN a

// match two related nodes, extract information from  the relationship
MATCH (a)-[r]->(b)
RETURN a, type(r), b

// extensive matching (node labels, relationship types, node/rel parameters)
MATCH (j: Person {name: “John”}) -[r:LIKES {level: 10}]-> (b: Beer {name: “Guiness”})
RETURN j.name, r.level, b.name

// can also use WHERE clause to narrow the matching
// eg.  WHERE … [ AND |  AND NOT | OR ]
//       WHERE value IN [ “a”, “b”, “c” ]
MATCH (j: Person) -[r:LIKES ]-> (b: Beer )
WHERE j.name = “John” AND r.level = 10 AND b.name = “Guiness”
RETURN j.name, r.level, b.name
```

Exercise: Find all Persons that contributed to Matrix



HINT:

match ….
return **distinct** ….

Exercise: Find all Persons that contributed to Matrix

```
match (p:Person)-[ ]->(m:Movie)
where m.title = "The Matrix"
return distinct p
```

Or

```
match (p:Person)-[ ]->(m:Movie {m.title = "The Matrix"} )
return distinct p
```

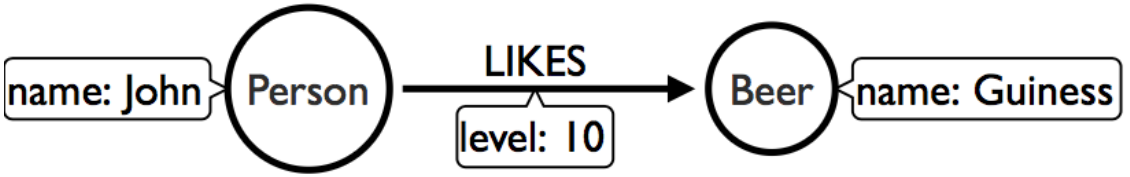
Introduction to Cypher

- Use aggregation functions to calculate a value on a set of data:
  - avg()
  - count()
  - max()
  - min()
  - sum()
  - collect()

```
// find the number if elements that contributed to “The Matrix”
match (p:Person)-[ ]->(m:Movie {m.title = "The Matrix"} )
return count ( distinct p )

// collect as a list the names of elements that contributed to “The Matrix”
match (p:Person)-[ ]->(m:Movie {m.title = "The Matrix"} )
return collect ( distinct p.name )
```

Introduction to Cypher

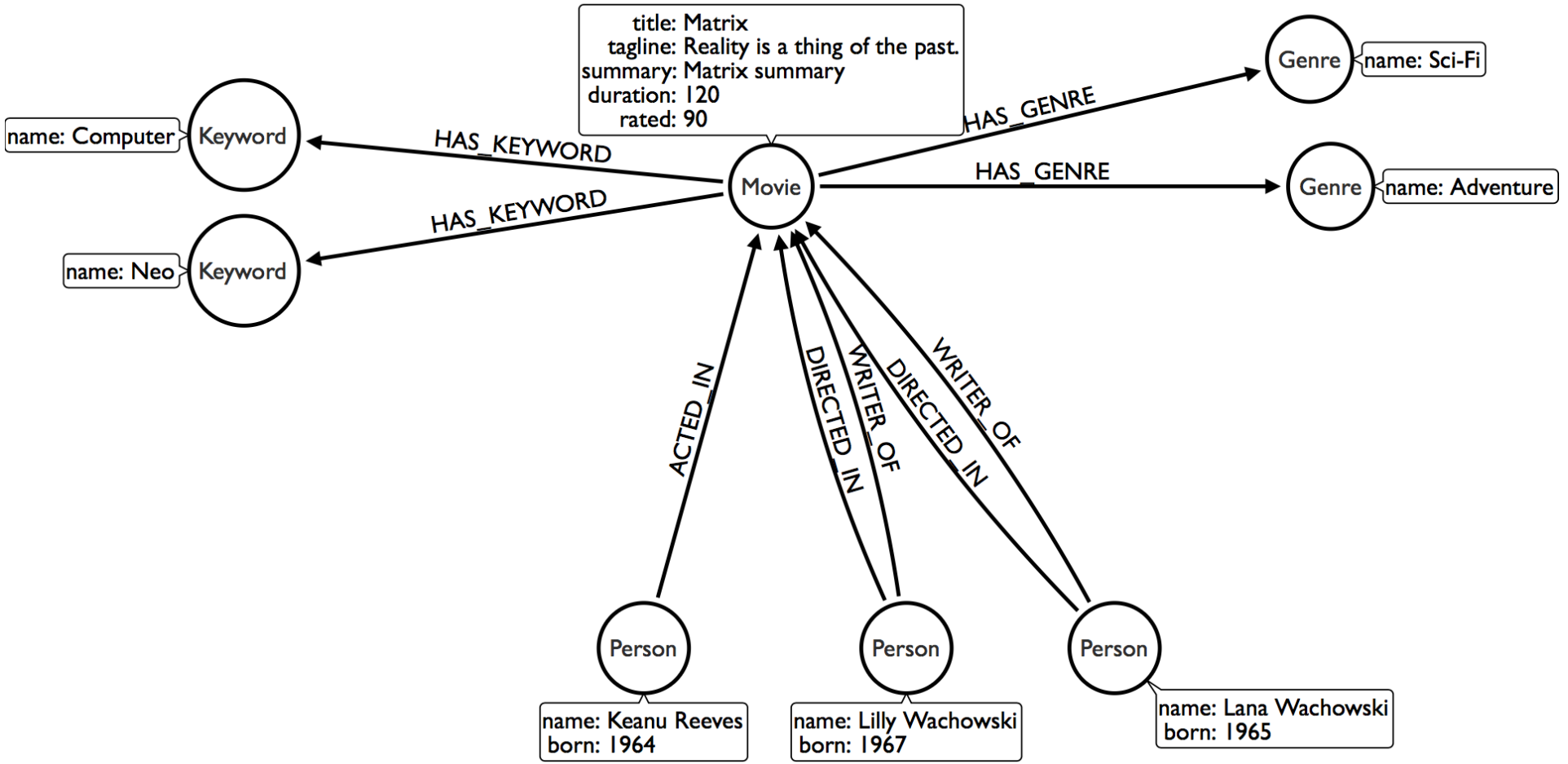
- Matching optional patterns on a graph



Acts same as **Match** only that it returns 'null' if no patterns are matched.

```
// match any two related nodes, direction not significant
OPTIONAL MATCH (a)--(b)
RETURN a,b

// match any two related nodes, direction is significant
OPTIONAL MATCH (a)-->(b)
RETURN a,b

//Same as above, we are only interested in start node of the relationship
OPTIONAL MATCH (a)-->()
RETURN a

// match two related nodes, extract information from  the relationship
OPTIONAL MATCH (a)-[r]->(b)
RETURN a, type(r), b

// extensive matching (node labels, relationship types, node/rel parameters)
OPTIONAL MATCH (j: Person {name: “John”}) -[r:LIKES {level: 10}]-> (b: Beer {name: “Guiness”})
RETURN j.name, r.level, b.name
```

Exercise: Find wether Tom Cruise has ever played in a movie with keyword Neo



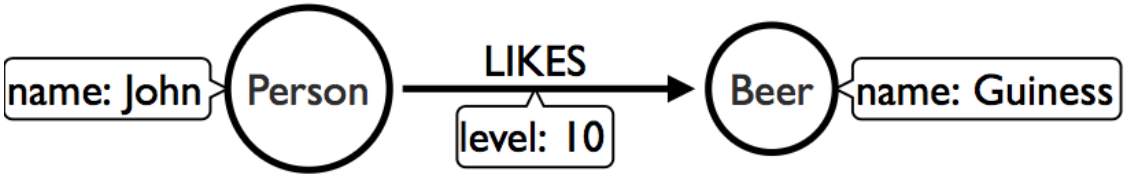
Exercise: Find wether Tom Cruise has ever played in a movie with keyword Neo

```
optional match (p:Person {name: "Tom Cruise"})-[]->(m:Movie)
where m.title = "The Matrix"
return p
```

Try without optional too. What is the difference ?

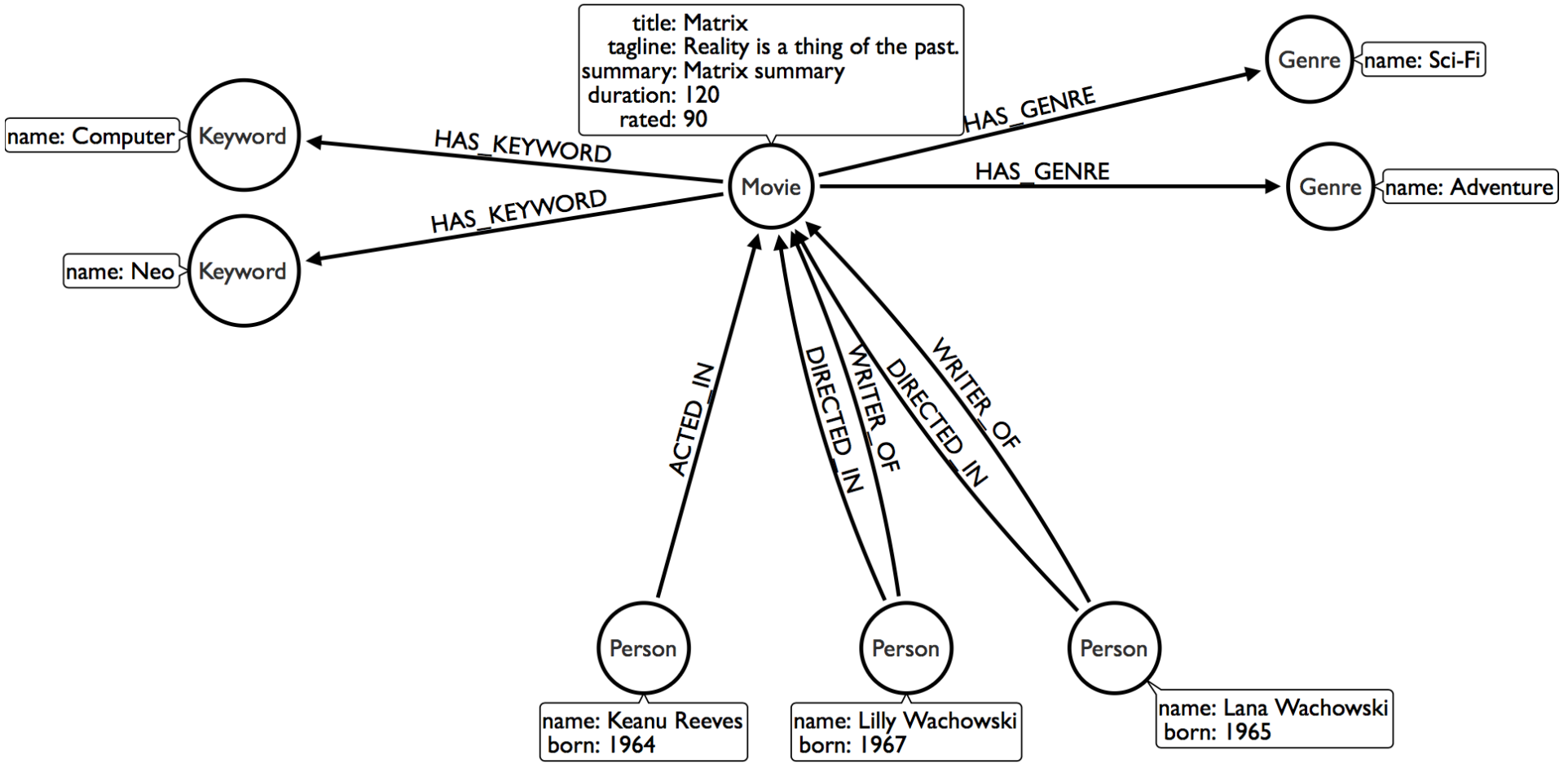
Introduction to Cypher

- Adding/updating properties




```
// add surname property
// the same way we could update it if there was already a surname property
MATCH (john: Person)
WHERE john.name=”John”
SET john.surname=”Smith”
RETURN john


// or
MATCH (john: Person {name: “John”})
SET john.surname=”Smith”
RETURN john
```

Exercise: Update the tagline of the movie to "Be afraid of the future"

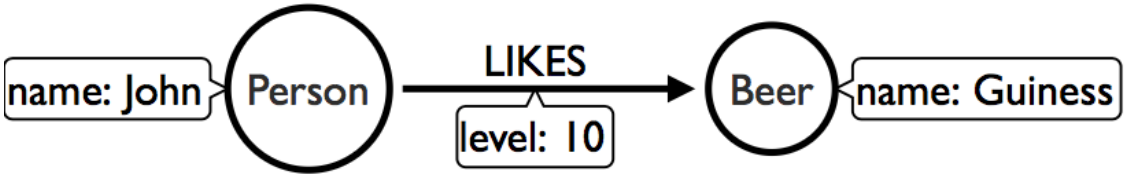


Exercise: Update the tagline of the movie to “Be afraid of the future”

```
match (m:Movie)
where m.title = "The Matrix"
set m.tagline="Be afraid of the future"
```

Introduction to Cypher

- Creating relationships




```
// add surname property
// the same way we could update it if there was already a surname property
MATCH (john: Person {name: “John”}), (beer: Beer {name: “Guiness”})
CREATE (john)-[:LOVES]->(beer)


// or
MATCH (john: Person {name: “John”}), (beer: Beer {name: “Guiness”})
MERGE (john)-[:LOVES]->(beer)
```

BUT MERGE is actually a more powerful mechanism !

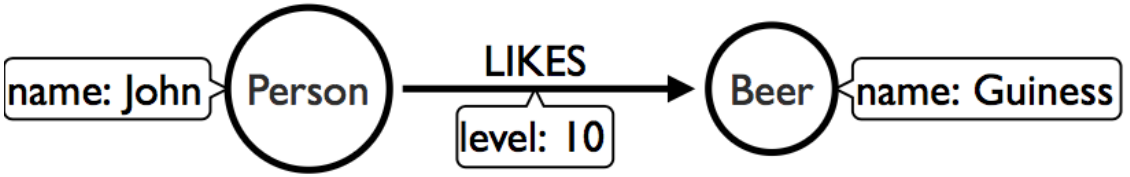
Exercise

- Create a new person first, and then assign to it the role of actor in the Matrix movie

```
CREATE (p:Person {name: “Dimitrios”})
MERGE (p)-[:ACTED_IN]-(:Movie {name: ”The Matrix”})
```

Introduction to Cypher

- Merge combines Match and Create




```
MERGE (john: Person {name: “John”})-[rel:LIKES]->(beer: Beer {name: “Guiness”})
ON MATCH  SET r.matched= r.matched + 1
ON CREATE SET r.matched=0
RETURN rel.matched AS numOfMatches
```

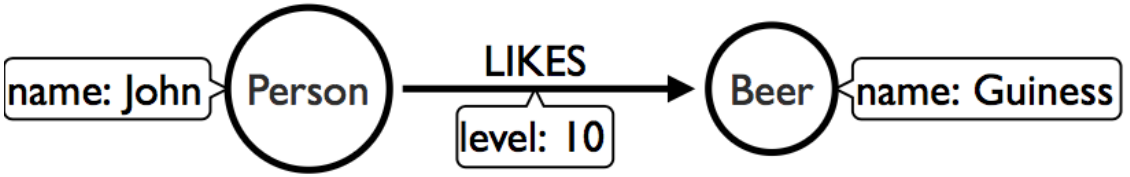
Exercise

Return the name of the producer of “Matrix”, if none exists then create one with your name.

```
MERGE (p: Person)-[:PRODUCED]-> (:Movie {name: “The Matrix”})
ON CREATE SET p.name = “Dimitrios”, p.born=1987, p.id=”5000”
RETURN p
```

Introduction to Cypher

- Deleting from graph




```
// Deleting nodes
// only if there are no relationships attached to it
MATCH (john: Person {name: “John”})
DELETE john

// Deleting relationships
MATCH (john: Person {name: “John”})-[rel:LIKES]->(beer: Beer {name: “Guiness”})
DELETE rel

// Deleting nodes and optionally relationships
MATCH (john: Person {name: “John”})
DETACH DELETE john

// Deleting all nodes and relationships
MATCH (n)
DETACH DELETE n
```

Exercise

- Delete yourself from the graph

```
MATCH (p: Person {name: “Dimitrios”})
DELETE p
```

Introduction to Cypher

- Matching multiple relationships

```
// Create a KNOWS relationship between the elements of the same film
MATCH  (a:Person)-[ :ACTED_IN|:DIRECTED]->()<-[:ACTED_IN|:DIRECTED]-(b:Person)
MERGE (a)-[:KNOWS]-(b)
```

- Variable length paths

```
// return all nodes that are reachable from (a), direction of relationship here is important
MATCH  (a)-[ *]→(b: LABEL) RETTURN a

// return all nodes that are two steps away from (a), direction here is insignificant
MATCH  (a)-[ *2]-(b) RETURN b

// [1..*] → more than one
// [2..3] → 2 to 3
// [..3] → up to 3
```

Exercise

- Find the friends of friends of Keanu Reeves

```
MATCH (p: Person {name: "Keanu Reeves"})-[:KNOWS*2]->(fof)
WHERE fof <> p
RETURN DISTINCT fof.name
```

Introduction to Cypher

- Using “WITH” to pass parameters to following query parts / patterns

  Is also used to manipulate output before passing it to the next query parts

  Example 1:

```
// Find the movies that the friends of friends of Keanu Reeves have played
MATCH (p: Person {name: "Keanu Reeves"})-[:KNOWS*2]->(fof)
WHERE fof <> p
WITH DISTINCT fof
MATCH (fof)-[:ACTED_IN]-(m)
RETURN distinct m
```

  Example 2:

```
// Return the maximum Person id
MATCH (n:Person) // match the pattern
WITH toInt(n.id) as intID // process the variable with “with”
RETURN MAX(intID) // pass it to the max function
```

Indexes

- Database indexes are used to make data retrieval faster
  - This comes with the overhead of more storage space being occupied and slower writes

- Two types of indexes:
  - Single property index:

    an index created on a single property for any given label

```
CREATE INDEX ON :Person(name)  // create
DROP    INDEX ON :Person(name)  // delete
```

  - Composite index:

    an index created on more than one properties for a given label

```
CREATE INDEX ON :Person(name, age)  // create
DROP    INDEX ON :Person(name, age)  // delete
```

Indexes

- Neo4j takes care of updating the index according to the database changes
- If the index is created on existing data then it might take some time before being available
- Usually there is no need to specify which index to use, cypher can figure it out depending on the properties and labels of the query.
  - If needed can specify it though (USING keyword)

Indexes

- Use single-property and composite index with WHERE using equality

```
// having a single-property index on name property of Person
MATCH (p:Person)
WHERE p.name = 'Keanu Reeves'
RETURN

// having a composite index on name and age properties of Person
MATCH (p:Person)
WHERE p.name = 'Keanu Reeves' AND p.age = 50
RETURN p
```

Indexes

- Index usage only for single-property indexes

```
- Range comparisons (only for single property indexes)
MATCH (n:Person)
WHERE n.age > 30
RETURN n

- Use index with IN (only for single property indexes)
MATCH (p:Person)
WHERE p.name IN ['Keanu Reeves', 'Tom Hanks']
RETURN p

- Use index with STARTS WITH / ENDS WITH / CONTAINS (only for single property
indexes)
MATCH (p:Person)
WHERE p.name STARTS WITH 'Tom'
RETURN p

- Use index when checking for the existence of a property
MATCH (p:Person)
WHERE exists(p.name)
RETURN p
```

Constraints

- Supported constraints:
  - Unique node property constraints
  - Node property existence constraints (EE only)
  - Relationship property existence constraints (EE only)
  - Node Keys (EE only)

- Constraints can apply before or after data population:
  - If they apply before, it is necessary that future data comply with them
  - If they apply on existing dataset, then they can apply only if the dataset already complies with them

Constraints

- Unique node property constraints
  - No more that one node with a specific label and one property value.
  - That property is not necessary to exist in the set of that node's properties.
  - Adding a unique property constraint on a property will also add a single-property index on that property.

```
// each person should have a unique name
CREATE CONSTRAINT ON (p: Person) ASSERT p.name IS UNIQUE

// delete uniqueness constraint on persons
DROP CONSTRAINT ON (p: Person) ASSERT p.name IS UNIQUE
```

Constraints

- Node property existence constraints
  - All nodes of a specific label should have certain property(-ies)

```
// each person should have a name property
CREATE CONSTRAINT ON (p: Person) ASSERT EXISTS(p.name)

// drop constraint
DROP CONSTRAINT ON (p: Person) ASSERT EXISTS(p.name)
```

Constraints

- Relationship property existence constraints
  - All relationships of a specific type have certain property(-ies)

```
// each ACTED_IN relationship should have a role property
CREATE CONSTRAINT ON ()-[acted :ACTED_IN]-() EXISTS(acted.role)

// drop constraint
DROP CONSTRAINT ON ()-[acted :ACTED_IN]-() EXISTS(acted.role)
```

Constraints

- Node Keys
  - All nodes of a specific label have a set of defined properties whose combination of values is unique and are all present in the properties list of that label.

```
// name and birth_date properties are node keys for Person nodes
CREATE CONSTRAINT ON (p :Person) ASSERT (p.name, p.birth_date) IS NODE KEY

// drop constraint
DROP CONSTRAINT ON (p :Person) ASSERT (p.name, p.birth_data) IS NODE KEY
```