

Graph Databases

UA.DETI.CBD

José Luis Oliveira / Carlos Costa

Some theory about graph theory

Graph Databases

- ❖ **Data:** a set of entities and their relationships
 - e.g., social networks, travelling routes, ...
 - We need to efficiently represent graphs
- ❖ **Basic operations:** finding the neighbours of a node, checking if two nodes are connected by an edge, updating the graph structure, ...
 - We need efficient graph operations
- ❖ $G = (V, E)$ is commonly modelled as
 - set of **nodes** (vertices) V
 - set of **edges** E
 - $n = |V|, m = |E|$
- ❖ Which data structure should be used?

Adjacency Matrix

❖ Bi-dimensional **array** A of $n \times n$ Boolean values

- Indexes = node identifiers
- A_{ij} indicates whether the two nodes i, j are connected

❖ **Pros:**

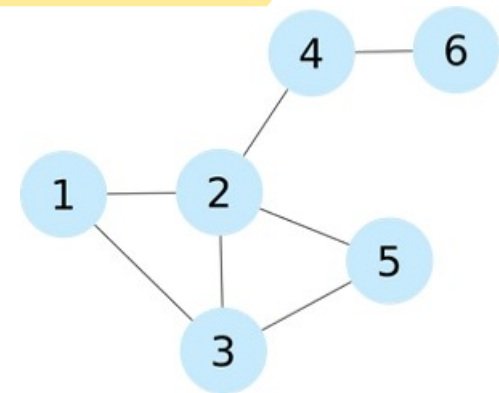
- Checking if two nodes are connected
- Adding/removing edges

❖ **Cons:**

- Quadratic space with respect to n
 - We usually have sparse graphs (lots of 0)
- Addition of nodes is expensive
- Retrieval of all the neighbouring - $O(n)$

❖ Other variants:

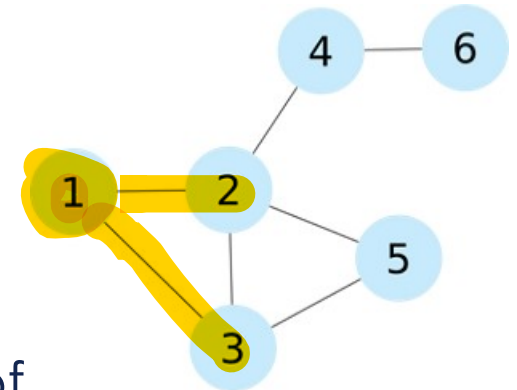
- Directed graphs, Weighted graphs, ...



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Adjacency List

- ❖ A **set of lists** where each accounts for the neighbours of one node
 - A vector of n pointers to adjacency lists
- ❖ Undirected graph:
 - An edge connects nodes i and $j \Rightarrow$ the list of neighbours of i contains the node j and vice versa
- ❖ **Pros:**
 - Obtaining the neighbours of a node
 - Cheap addition of nodes to the structure
 - Compact representation of sparse matrices
- ❖ **Cons:**
 - Checking an edge between two nodes



$N1 \rightarrow \{N2, N3\}$

$N2 \rightarrow \{N1, N3, N5\}$

$N3 \rightarrow \{N1, N2, N5\}$

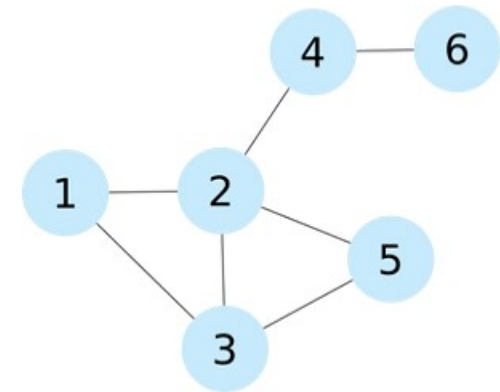
$N4 \rightarrow \{N2, N6\}$

$N5 \rightarrow \{N2, N3\}$

$N6 \rightarrow \{N4\}$

Incidence Matrix

- ❖ Bi-dimensional Boolean matrix of n rows and m columns
 - A **column** represents an **edge**
 - Nodes that are connected by a certain edge
 - A **row** represents a **node**
 - All edges that are connected to the node
- ❖ **Pros:**
 - For representing hypergraphs, where one edge connects an arbitrary number of nodes
- ❖ **Cons:**
 - Requires $n \times m$ bits

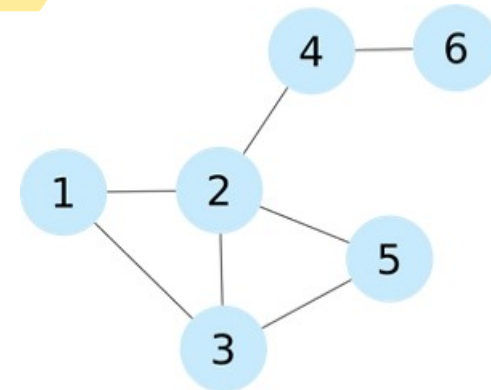


$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Laplacian Matrix

❖ Bi-dimensional **array** of $n \times n$ integers

- Diagonal of the Laplacian matrix indicates the **degree** of the node
- The rest of positions are set to -1 if the two vertices are connected, 0 otherwise



❖ **Pros:**

- Allows analyzing the graph structure by means of spectral analysis
 - Calculates the eigenvalues

❖ **Cons:**

- = Adjacency Matrix

$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 3 & 0 & -1 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ 0 & -1 & -1 & 0 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

Graph Traversals

- ❖ Single step **traversal** from element i to element j , where $i, j \in (V \cup E)$
- ❖ Expose explicit **adjacencies** in the graph
 - e_{out} : traverse to the outgoing edges of the vertices
 - e_{in} : traverse to the incoming edges of the vertices
 - v_{out} : traverse to the outgoing vertices of the edges
 - v_{in} : traverse to the incoming vertices of the edges
 - e_{lab} : allow (or filter) all edges with the label
 - \in : get element property values for key r
 - e_p : allow (or filter) all elements with the property s for key r
 - $\in =$: allow (or filter) all elements that are the provided element

Graph Traversals

- ❖ Single step traversals can **compose complex traversals** of arbitrary length
 - e.g., find all friends of Alberto
 - Traverse to the outgoing edges of vertex i (representing Alberto), then only allow those edges with the label friend, then traverse to the incoming (i.e. head) vertices on those friend-labeled edges. Finally, of those vertices, return their name property.”

$$f(i) = (\epsilon^{name} \circ v_{in} \circ e_{lab}^{friend} \circ e_{out})(i)$$

Types of Graphs

❖ Single-relational

- Edges are homogeneous in meaning
 - e.g., all edges represent friendship

❖ Multi-relational (property) graphs

- Edges are typed or labeled
 - e.g., friendship, business, communication
- Vertices and edges in a property graph maintain a set of key/value pairs
 - Representation of non-graphical data (properties)
 - e.g., name of a vertex, the weight of an edge

Graph Databases

- ❖ A graph database = a set of graphs
- ❖ Types of graphs:
 - Directed-labeled graphs
 - e.g., XML, RDF, traffic networks
 - Undirected-labeled graphs
 - e.g., social networks, chemical compounds
- ❖ Types of graph databases:
 - Non-transactional = few numbers of very large graphs
 - e.g., Web graph, social networks, ...
 - Transactional = large set of small graphs
 - e.g., chemical compounds, biological pathways, linguistic trees each representing the structure of a sentence...

Transactional Graph Databases

Types of Queries

❖ **Sub-graph** queries

- Searches for a specific pattern in the graph database
- A small graph or a graph, where some parts are uncertain
 - e.g., vertices with wildcard labels
- More general type: sub-graph isomorphism

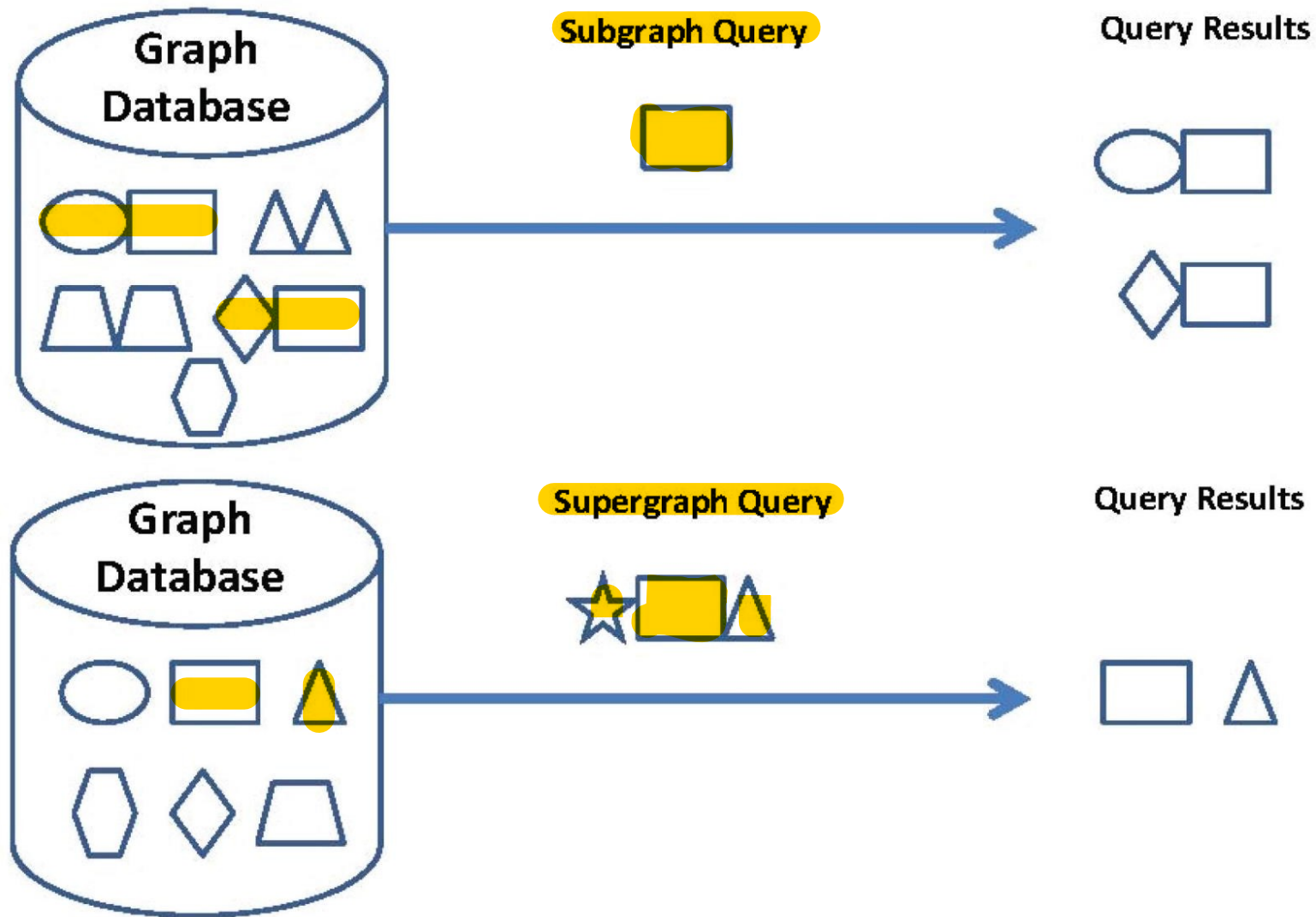
❖ **Super-graph** queries

- Searches for the graph database members of which their whole structures are contained in the input query

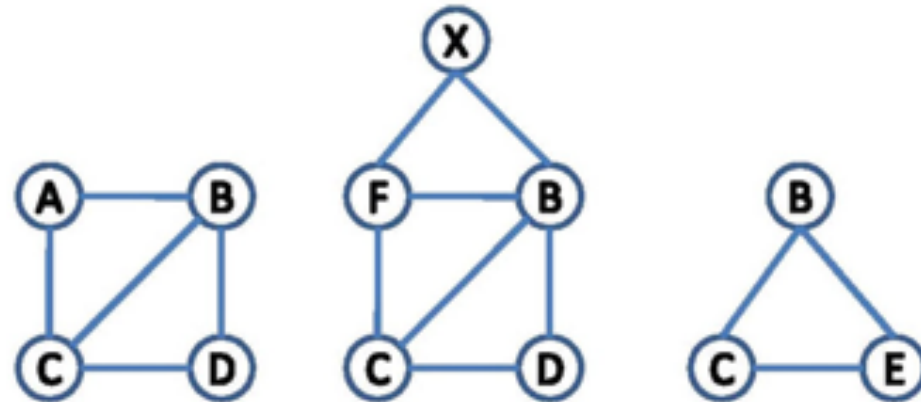
❖ **Similarity** (approximate matching) queries

- Finds graphs which are similar, but not necessarily isomorphic to

Graph queries



Graph queries



sub-graph:

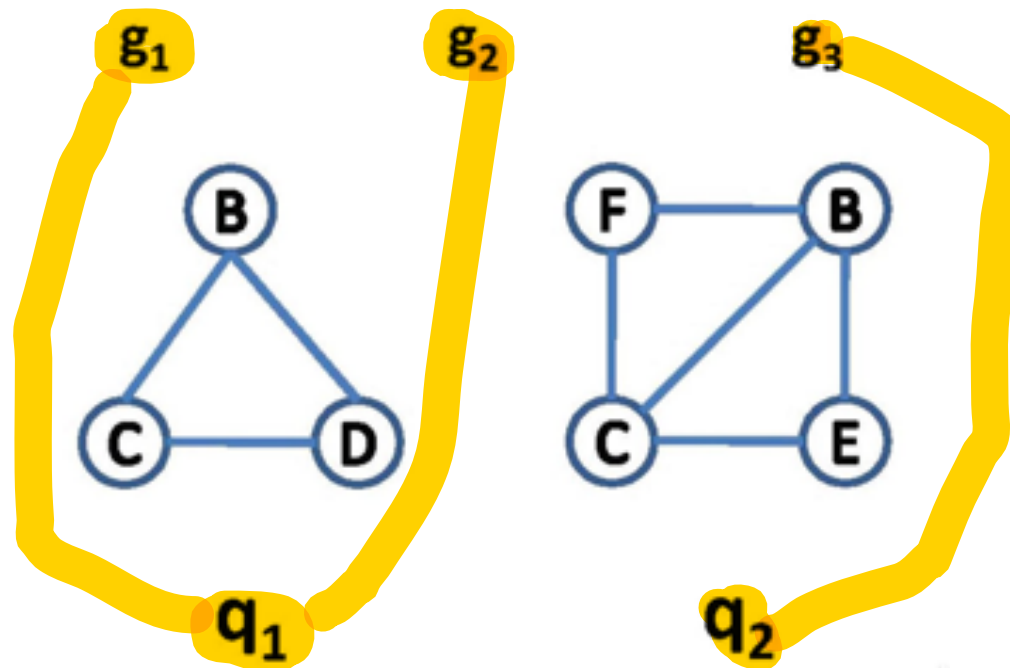
$q_1: g_1, g_2$

$q_2: \emptyset$

super-graph:

$q_1: \emptyset$

$q_2: g_3$



Sub-graph Query Processing

Mining-Based Graph Indexing Techniques

- ❖ Idea: if features of query graph q do not exist in data graph G , then G cannot contain q as its sub-graph
- ❖ Graph-mining methods extract selected features (sub-structures) from the graph database members
 - An inverted index is created for each feature
- ❖ Answering a sub-graph query q :
 - Identifying the set of features of q
 - Using the inverted index to retrieve all graphs that contain the same features of q

Sub-graph Query Processing

Non Mining-Based Graph Indexing Techniques

- ❖ Focus on indexing whole constructs of the graph database
 - Instead of indexing only some selected features
- ❖ Cons:
 - Can be less effective in their pruning (filtering) power
 - May need to conduct expensive structure comparisons in the filtering process
- ❖ Pros:
 - Can handle graph updates with less cost
 - Do not rely on the effectiveness of the selected features
 - Do not need to rebuild whole indexes

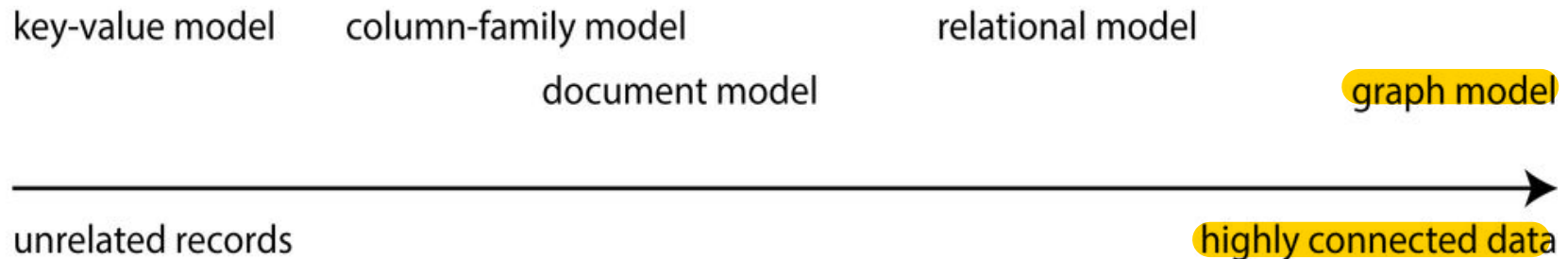
Graph Similarity Queries

- ❖ Find sub-graphs in the database that are *similar* to query q
 - Allows for node mismatches, node gaps, structural differences, ...
- ❖ Usage: when graph databases are noisy or incomplete
 - Approximate graph matching query-processing techniques can be more useful and effective than exact matching

Graph-oriented Database

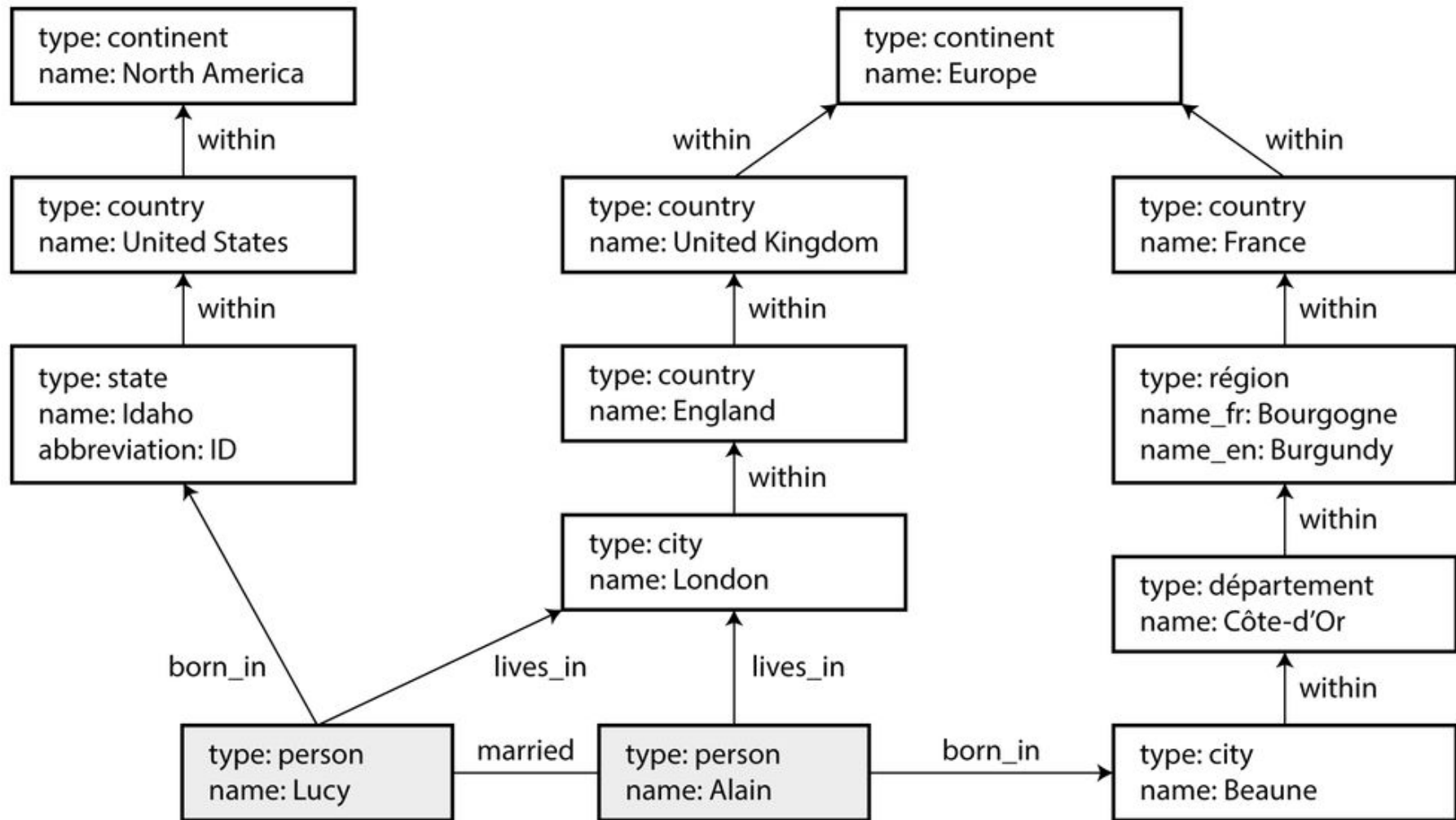
Graph-like Data Models

- ❖ Many-to-many relationships are an important distinguishing feature between different data models.



- ❖ The relational model can handle simple cases of many-to-many relationships, but
 - as the connections become more complex, it becomes more natural to start modeling as a graph.

Graph-like Data Models

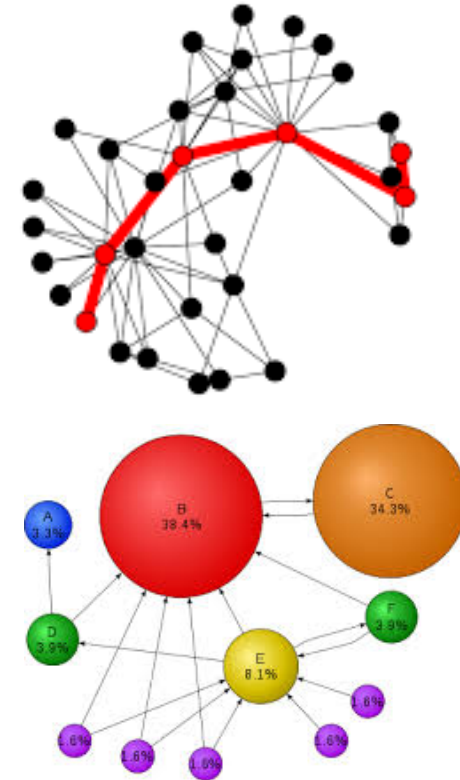


Graph-like Data Models

- ❖ A graph consists of two kinds of object:
 - **vertices** (also known as nodes or entities)
 - **edges** (also known as relationships).
- ❖ Many kinds of data can be modelled as a graph:
 - **Social graphs** – vertices are people, edges indicate which people know each other.
 - The **web graph** – vertices are web pages, edges indicate HTML links to other pages.
 - Road or rail **networks** – vertices are junctions, and edges represent the roads or railway lines between them.

Graph-like Data Models

- ❖ Well-known **algorithms** can operate on these graphs:
for example,
 - the **shortest path** in a road network is useful for routing.
 - **PageRank** on the web graph to determine the popularity of a web page.
 - Closeness, betweenness, etc.



Graph-like Data Models

- ❖ There are several different, but related, ways of **structuring** and **querying** data in graphs. Two examples:
 - **property graph** model
 - implemented by Neo4j, Titan, InfiniteGraph
 - the **triple-store** model
 - implemented by Datomic, AllegroGraph and others.
- ❖ Some **declarative query languages** for graphs
 - Cypher
 - SPARQL
 - Datalog

Property graphs

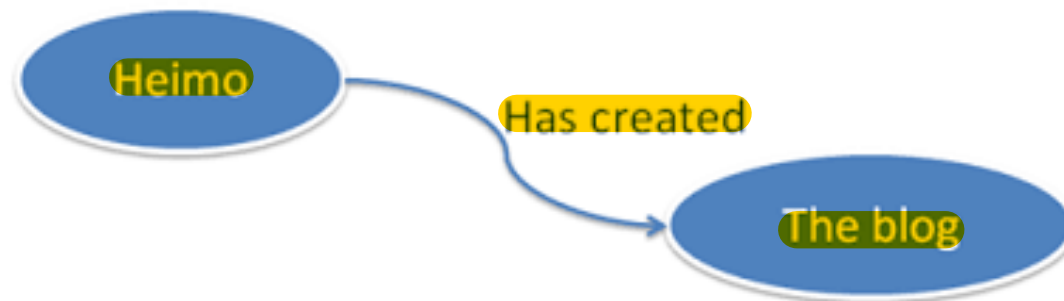
- ❖ Each **vertex** consists of:
 - a unique identifier,
 - a set of outgoing edges,
 - a set of incoming edges, and
 - a collection of properties (key-value pairs).
- ❖ Each **edge** consists of:
 - a unique identifier,
 - the vertex at which the edge starts (the tail vertex),
 - the vertex at which the edge ends (the head vertex),
 - a label to describe the type of relationship between the two vertices, and
 - a collection of properties (key-value pairs).

Property graphs

- ❖ Any vertex can have an edge connecting it with any other vertex.
 - There is no schema that restricts which kinds of things can or cannot be associated.
- ❖ Given any vertex,
 - We can efficiently find both incoming and outgoing edges.
 - Traverse the graph.
- ❖ Different labels for different kinds of relationship
 - Allow storing several different kinds of information in a single graph, while still maintaining a clean data model.

Triple-stores

- ❖ The **triple-store model** is mostly equivalent to the property graph model
 - using different words to describe the same ideas.
- ❖ Information is stored in the form of very simple three-part statements:
 - **subject, predicate, object.**



Triple-stores

- ❖ The **subject** of a triple is equivalent to a vertex in a graph.
- ❖ The **object** is one of two things:
 - a value in a primitive datatype, such as a string or a number.
 - In that case, the **predicate** and object of the triple are equivalent to the key and value of a property on the subject vertex.
 - For example, (lucy, age, 33) is like a vertex lucy with properties {"age":33}.
 - another vertex in the graph.
 - In that case, the **predicate** is an edge in the graph, the subject is the tail vertex and the object is the head vertex.
 - For example, in (lucy, marriedTo, alain).

Triples examples

- ❖ Using *Turtle*, a format that is a subset of *Notation3* (N3).

@prefix : <urn:x-example:>.

_:lucy a :Person; :name "Lucy"; :bornIn _:idaho.

_:idaho a :Location; :name "Idaho"; :type "state"; :within _:usa.

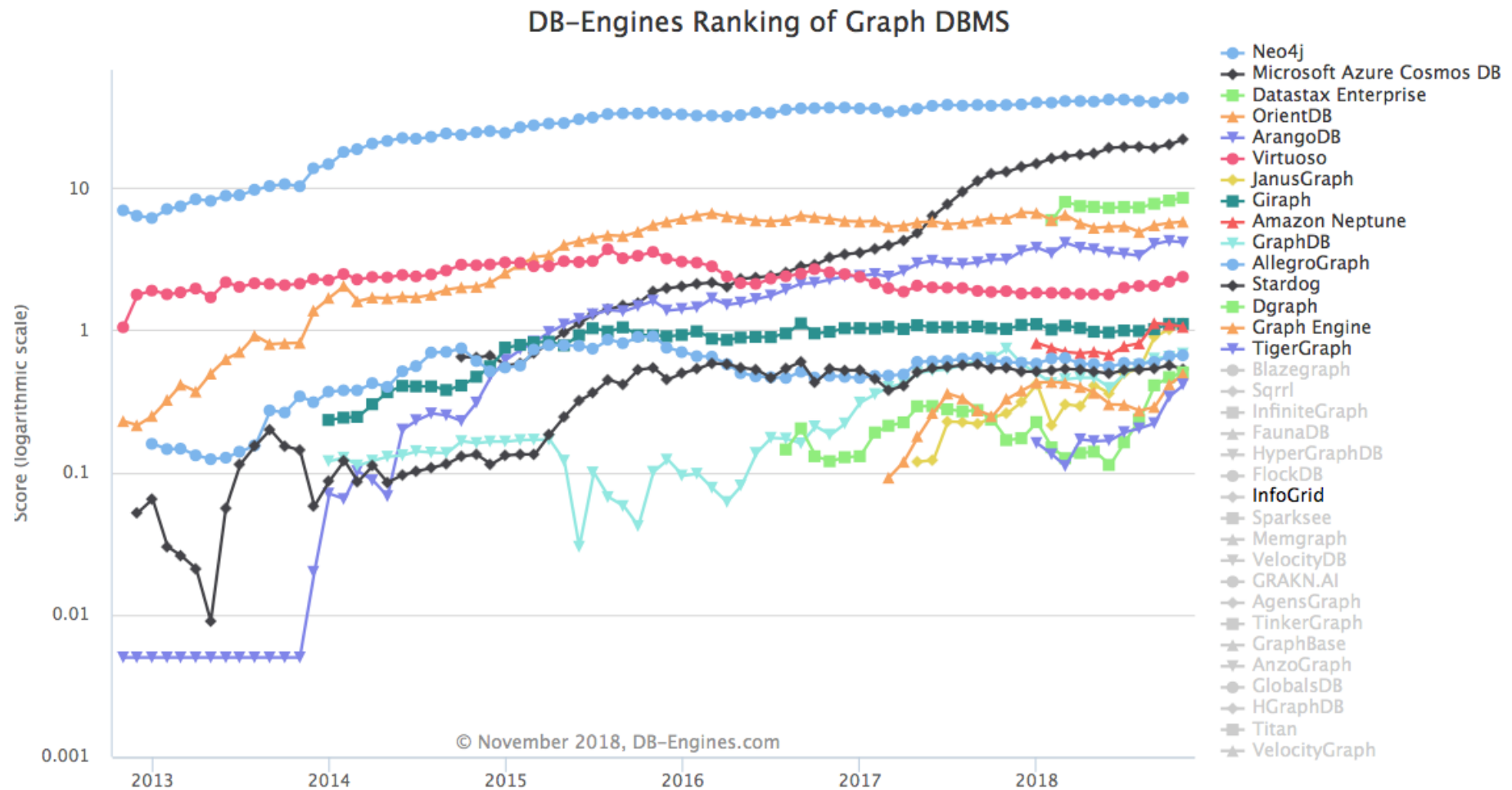
_:usa a :Location; :name "United States"; :type "country"; :within _:namerica.

_:namerica a :Location; :name "North America"; :type "continent".

Graph Databases



Graph Databases Popularity



Graph Databases

❖ Query patterns

- Create, update or remove a node / relationship in a graph
- Graph algorithms (shortest paths, spanning trees, ...)
- General graph traversals
- Sub-graph queries or super-graph queries
- Similarity based queries (approximate matching)

❖ Representatives

- Neo4j, Titan, Apache Giraph, InfiniteGraph, FlockDB

❖ Multi-model

- OrientDB, OpenLink Virtuoso, ArangoDB

Graph Databases

❖ Suitable use cases

- Social networks, routing, dispatch, and location-based services,
- recommendation engines, chemical compounds, biological pathways, linguistic trees, ...
- i.e. simply for graph structures

❖ When not to use

- Extensive batch operations are required
 - Multiple nodes / relationships are to be affected
- Only too large graphs to be stored
 - Graph distribution is difficult or impossible at all

Neo4j Graph Database



- ❖ Graph database
 - <https://neo4j.com/>
- ❖ Features
 - Open source, massively scalable (billions of nodes), high availability, fault-tolerant, master-slave replication, ACID transactions, embeddable, ...
 - Expressive graph query language (Cypher), traversal framework
- ❖ Developed by Neo Technology
- ❖ Implemented in Java
- ❖ Operating systems: cross-platform
- ❖ Initial release in 2007

Features of Neo4j

❖ Data model (flexible schema)

- Neo4j follows a data model named native property graph model.
- The graph contains **nodes** (entities) and these nodes are connected with each other (depicted by **relationships**). Nodes and relationships store data in key-value pairs known as **properties**.
- In Neo4j, there is no need to follow a fixed schema.

❖ ACID properties

- Neo4j supports full ACID (Atomicity, Consistency, Isolation, and Durability) rules.

Features of Neo4j

❖ Scalability and reliability

- You can scale the database by increasing the number of reads/writes, and the volume without affecting the query processing speed and data integrity.
- Neo4j also provides support for **replication** for data safety and reliability.

❖ Cypher Query Language

- Neo4j provides a powerful declarative query language known as Cypher.
- It uses ASCII-art for depicting graphs.
- Cypher is easy to learn and can be used to create and retrieve relations between data without using the complex queries like Joins.

Features of Neo4j

❖ Built-in web application

- Neo4j provides a built-in **Neo4j Browser** web application. Using this, you can create and query your graph data.

❖ Drivers – Neo4j can work with

- REST API to work with programming languages such as Java, Spring, Scala etc.
- Java Script to work with UI MVC frameworks such as Node JS.
- It supports two kinds of Java API: Cypher API and Native Java API to develop Java applications.

❖ Indexing – Neo4j supports Indexes by using Apache Lucene.

Data Model

- ❖ Database system structure
 - **Instance** → **single graph**
- ❖ Property graph = directed labeled multigraph
 - Collection of vertices (nodes) and edges (relationships)
- ❖ Graph **node**
 - Has a **unique** (internal) identifier
 - Can be associated with a **set of labels**
 - Allow us to **categorize nodes**
 - Can also be associated with a **set of properties**
 - Allow us to **store additional data together with nodes**

Data Model

❖ Graph **relationship**

- Has a **unique** (internal) identifier
- Has a **direction**
 - Relationships are equally well traversed in either direction!
 - Directions can be ignored when querying
- Always has a **start** and **end** node
 - Can be **recursive** (i.e. loops are allowed)
- Is **associated** with **exactly one type**
- Can also be **associated** with a **set of properties**

Data Model

❖ Node and relationship **properties**

- Key-value pair
 - Key is a string
 - Value is an atomic value of any primitive data type, or an array of atomic values of one primitive data type

❖ Primitive **data types**

- **boolean** – boolean values **true** and **false**
- **byte, short, int, long** – integers (1B, 2B, 4B, 8B)
- **float, double** – floating-point numbers (4B, 8B)
- **char** – one Unicode character
- **String** – sequence of Unicode characters

Cypher

- ❖ Declarative graph query language
 - Allows for expressive and efficient querying and updates
 - Inspired by SQL (query clauses) and SPARQL (pattern matching)
- ❖ OpenCypher
 - Ongoing project aiming at Cypher standardization
<http://www.opencypher.org/>
- ❖ Clauses
 - E.g. MATCH, RETURN, CREATE, ...
 - Clauses are (almost arbitrarily) chained together
 - Intermediate result of one clause is passed to a subsequent one

Cypher – Nodes

❖ Cypher uses a pair of parentheses to represent Nodes

- Like a circle or a rectangle with rounded corners.

`()`

- Represents an anonymous, uncharacterized **node**.

`(matrix)`

- If we want to refer to the node elsewhere, we can add an variable

`(:Movie)`

- The **Movie label** declares the node's type or role

`(matrix:Movie)`

`(matrix:Movie {title: "The Matrix"})`

`(matrix:Movie {title: "The Matrix", released: 1999})`

- The node's **properties** (title, released, et cetera) are represented as a list of key/value pairs, enclosed within a pair of braces

`(matrix:Movie:Promoted)`



Cypher – Relationships

❖ Cypher uses arrows to represent relationships

`-, ->, <-`

- Relationships are arrows pointing from one node to another

`(node1)-[:REL_TYPE]->(node2)`

- General relation, from node1 to node2

`(actor:Person)-[:ACTED_IN]->(movie:Movie)`

- Retrieve all nodes that had a relationship type ACTED_IN with other nodes.

❖ Query examples

`MATCH (node1)-[rel:TYPE]->(node2)`

`RETURN rel.property`

- Generic format, from node to node2.

`MATCH (actor:Person)-[rel:ACTED_IN]->(movie:Movie)`

`RETURN rel.roles`

- Roles of actors that acted in any movie.

`MATCH (n)-->(m) RETURN n, m;`

- every pair of nodes with a relationship going from n to m.

Cypher – Patterns

- ❖ Combining the syntax for nodes and relationships, we can express patterns.

```
MATCH (matrix:Movie {title:"The Matrix"} )  
      <-[role:ACTED_IN {roles:["Neo"]}]-  
      (keanu:Person {name:"Keanu Reeves"})  
RETURN matrix, role, keanu
```



matrix

role

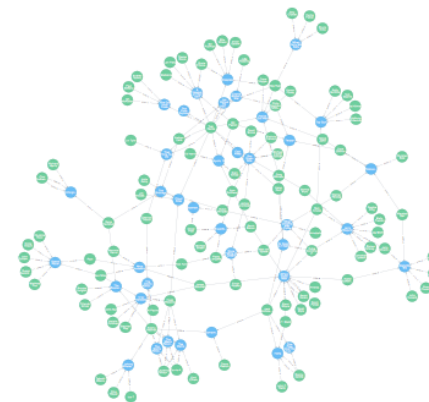
keanu

```
{  
  "title": "The Matrix",  
  "tagline": "Welcome to the Real  
World",  
  "released": 1999  
}
```

```
{  
  "roles": [  
    "Neo"  
  ]  
}
```

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

```
MATCH cast = (:Person)-[:ACTED_IN]->(:Movie)  
RETURN cast
```



Cypher – Selection

❖ MATCH

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

- all nodes

```
MATCH (me:Person) WHERE me.name="My Name" RETURN me.name
```

```
MATCH (me:Person {name:"My Name"}) RETURN me.name
```



```
MATCH (movie:Movie)
```

```
WHERE movie.title = "Mystic River"
```

```
SET movie.released = 2003
```

```
RETURN movie.title AS title, movie.released AS released
```

title	released
Mystic River	2003

Cypher – Filtering

❖ WHERE

```
MATCH (tom:Person)-[:ACTED_IN]->()-[:ACTED_IN]-(actor:Person)
WHERE tom.name="Tom Hanks" AND actor.born < tom.born
RETURN DISTINCT actor.name AS Name
```

- ??

```
MATCH (gene:Person)-[:ACTED_IN]->()-[:ACTED_IN]-(other:Person)
WHERE gene.name="Gene Hackman" AND exists( (other)-[:DIRECTED]->() )
RETURN DISTINCT other
```

- ??

```
MATCH (gene:Person {name:"Gene Hackman"})-[:ACTED_IN]->(movie:Movie),
      (other:Person)-[:ACTED_IN]->(movie),
      (robin:Person {name:"Robin Williams"})
WHERE NOT exists( (robin)-[:ACTED_IN]->(movie) )
RETURN DISTINCT other
```

- ??

Cypher – Ordering

❖ ORDER BY, LIMIT, SKYP, DISTINCT

- ❖ Return the five oldest people in the database

```
MATCH (person:Person)
```

```
RETURN person
```

```
ORDER BY person.born
```

- ❖ LIMIT 5;

- ❖ List of the oldest actors

```
MATCH (actor:Person)-[:ACTED_IN]->()
```

```
RETURN DISTINCT actor
```

```
ORDER BY actor.born
```

Variable Length Paths

MATCH (node1)-[*]-(node2)

- ❖ Relationships that traverse any depth are:
`(a)-[*]->(b)`
- ❖ Specific depth of relationships
`(a)-[*depth]->(b)`
- ❖ Relationships from one to four levels deep
`(a)-[*1..4]->(b)`
- ❖ Relationships of type KNOWS at 3 levels distance:
`(a)-[:KNOWS*3]->(b)`
- ❖ Relationships of type KNOWS or LIKES from 2 levels distance:
`(a)-[:KNOWS|:LIKES*2..]->(b)`

Indexes

- ❖ Neo4j doesn't use indexes to speed up JOINS
 - They are useful for finding the starting points by value, textual prefix or range
- ❖ To search efficiently people by name:
`CREATE INDEX ON :Person(name);`
- ❖ Now, the lookup of "Gene Hackman" will be faster
`MATCH (gene:Person)-[:ACTED_IN]->(movie),
 (other:Person)-[:ACTED_IN]->(movie)
WHERE gene.name="Gene Hackman"
RETURN DISTINCT other;`
- ❖ To remove the index:
`DROP INDEX ON :Person(name);`

Aggregation

- ❖ Cypher provides support for a number of aggregate functions
 - **count(x)** Count the number of occurrences
 - **min(x)** Get the lowest value
 - **max(x)** Get the highest value
 - **avg(x)** Get the average of a numeric value
 - **sum(x)** Sum up values
 - **collect(x)** Collect all the values into an collection

```
MATCH (person:Person)-[:ACTED_IN]->(movie:Movie)
RETURN person.name, count(movie)
ORDER BY count(movie) DESC
LIMIT 10;
```

- Top ten actors who acted in the most movies

Creating Nodes

❖ CREATE

```
CREATE (node:label { key1: value, key2: value, ... })
```

```
CREATE (Aveiro)
```

```
CREATE (Porto),(Coimbra),(Espinho)
```

```
CREATE (ric:person:player)
```

```
CREATE (leo:person:player)
```

```
CREATE (aveiro:cidade{name:"Aveiro"})
```

```
CREATE (ricg:player{name: "Ricardo Gomes",  
YOB: 1985, POB: "Porto"})
```

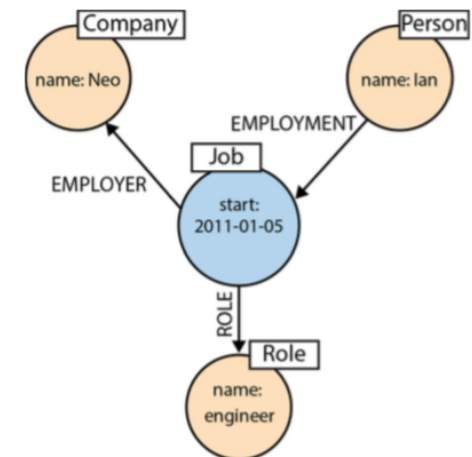


Creating Relationships

```
CREATE (RuiPatricio:player{name: "Rui Patrício", YOB: 1988, POB: "Leiria"})
CREATE (PT:Country {name: "Portugal"})
CREATE (RuiPatricio)-[r:Guarda_Redes]->(PT)
CREATE (RuiPatricio)-[:JogadorDeFutebol]->(PT)
RETURN RuiPatricio, PT
```



```
CREATE (:Person {name:'Ian'})-[:EMPLOYMENT]->
  (employment:Job
    {start_date:'2011-01-05'}) -[:EMPLOYER]->
    (:Company {name:'Neo'}), (employment)-[:ROLE]->
    (:Role {name:'engineer'})
```



Removing nodes/relationships

❖ DELETE

- Removes nodes, relationships or paths from the data graph
- Relationships must be removed before the nodes
 - Unless the DETACH modifier is specified

```
MATCH (p:Person {name:"My Name"})
```

```
DELETE p
```

- Remove node "My name". Error if it has relations

```
MATCH (me:Person {name:"My Name"})
```

```
OPTIONAL MATCH (me)-[r]-(C)
```

```
DELETE me,r
```

- Remove me (node "My name") and any relationships with "me"..

```
MATCH (n) DETACH DELETE n
```

- delete all nodes and relationships

Importing Data

❖ LOAD CSV

❖ Content of "movies.csv"

id,title,country,year

1,Wall Street,USA,1987

2,The American President,USA,1995

3,The Shawshank Redemption,USA,1994

❖ In Cypher:

LOAD CSV WITH HEADERS

FROM "http://neo4j.com/docs/stable/csv/intro/movies.csv"

AS line

CREATE (movie:Movie

{ id:line.id, title:line.title, released:toInt(line.year) });

Other Write Clauses

❖ SET clause

- allows to...
 - set a value for a property, or remove a property when NULL is assigned
 - replace all the current properties with new ones
 - add new properties to the existing ones
 - add labels to nodes
- Cannot be used to set relationship types

❖ REMOVE clause

- Allows to...
 - remove a particular property remove labels from nodes
 - Cannot be used to remove relationship types

Summary

- ❖ Graph theory
 - brief concepts
- ❖ Graph-oriented databases
 - Property graphs
- ❖ Neo4j graph database
- ❖ Cypher (graph query language)
 - Read (sub-)clauses: MATCH, WHERE, ...
 - Write (sub-)clauses: CREATE, DELETE, SET, REMOVE, ...
 - General (sub-)clauses: RETURN, WITH, ORDER BY, LIMIT, ...

Resources

- ❖ Pramod J Sadalage and Martin Fowler, **NoSQL Distilled**. Addison-Wesley, 2012
- ❖ Ian Robinson, Jim Webber and Emil Eifrem, **Graph Databases**, O'Reilly's, 2013
 - <https://neo4j.com/graph-databases-book/>
- ❖ Neo4j
 - <https://neo4j.com/developer/>
- ❖ Martin Svoboda, "B4M36DS2: Database Systems"
 - <http://www.ksi.mff.cuni.cz/~svoboda/courses/171-B4M36DS2/>