Data Management

Domenico Lembo, Maurizio Lenzerini

Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università di Roma "La Sapienza"

Part 3
NoSQL Databases: graph-oriented models

Academic Year 2024/2025

http://www.diag.uniroma1.it/~lenzerini/index.html/?q=node/53

The NoSQL movement

- Since the 80s, the dominant back end of business data systems has been a relational database.

- It’s remarkable that many architectural variations have been explored in the design of clients, front ends, and middleware, on a multitude of platforms and frameworks, but haven’t until 15 years ago questioned the architecture of the back end.

- In the past 15 years, we’ve been faced with data that are bigger in volume, change more rapidly, and are more structurally varied (in a definition, *Big Data*) than those typically dealt with by traditional RDBMS deployments.

- The NOSQL (Not Only SQL) movement has arisen in response to these challenges.

Limits of relational technologies for Big Data

- The schema of a relational database is static and has to be understood from the beginning of a database design => Big Data may change at an high rate over the time, so does their structure.

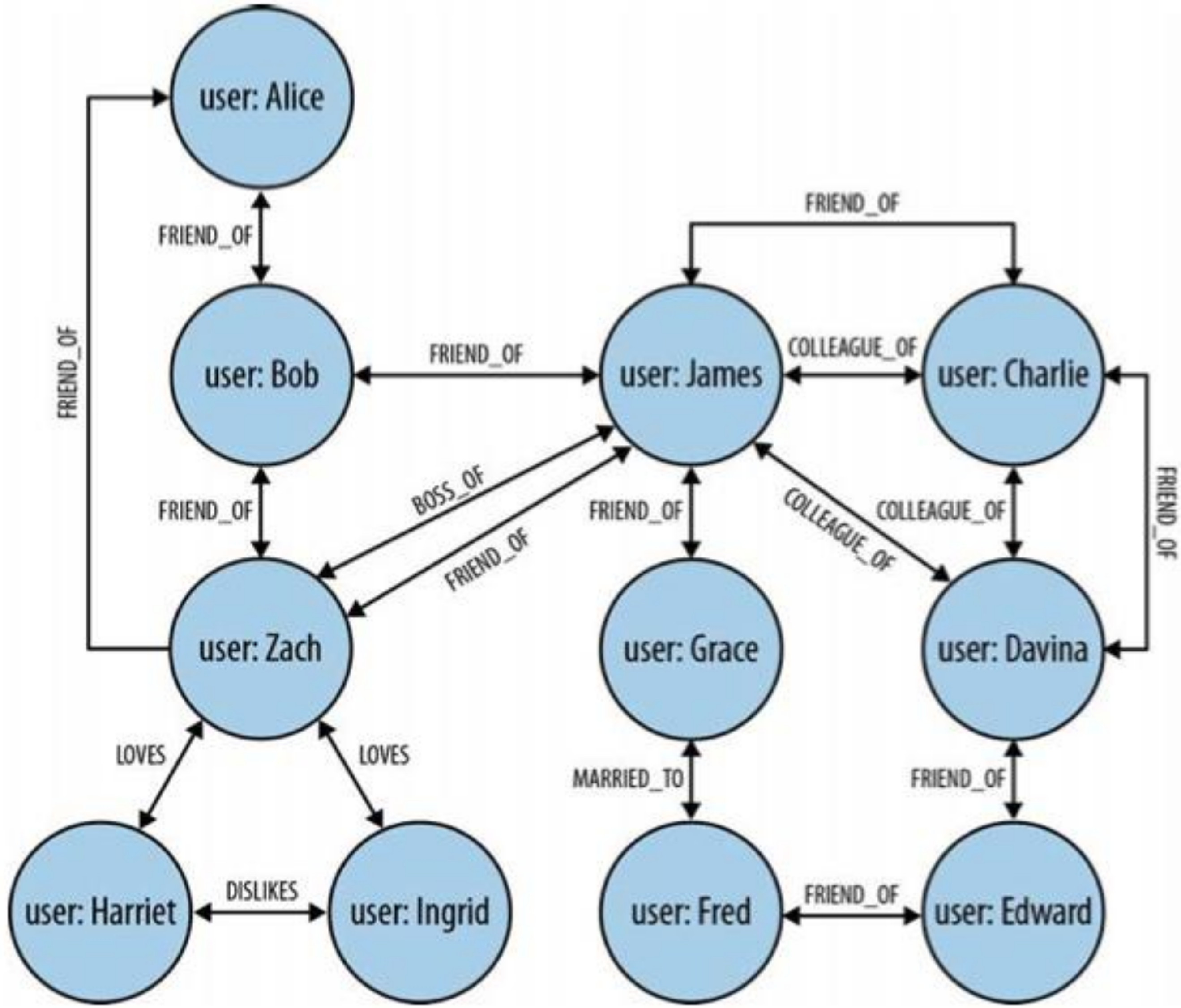
- Relational databases do not well behave in the presence of high variety in the data and => Big Data may be regularly or irregularly structured, dense or sparse, connected or disconnected.

- Query execution times increase as the size of relational tables and the number of joins grow (so-called *join pain*) => this is a problem in particular queries not seen yet.

Types of NoSQL data models

- Graph-based
- Key-value stores
- Document-based
- Column-oriented

Graph databases



Key-values stores

Key	Value
employee_1	name@Tom-surn@Smith-off@41-buil@A4-tel@45798
employee_2	name@John-surn@Doe-off@42-buil@B7-tel@12349
employee_3	name@Tom-surn@Smith
office_41	buil@A4-tel@45798
office_42	buil@B7-tel@12349

Document-based

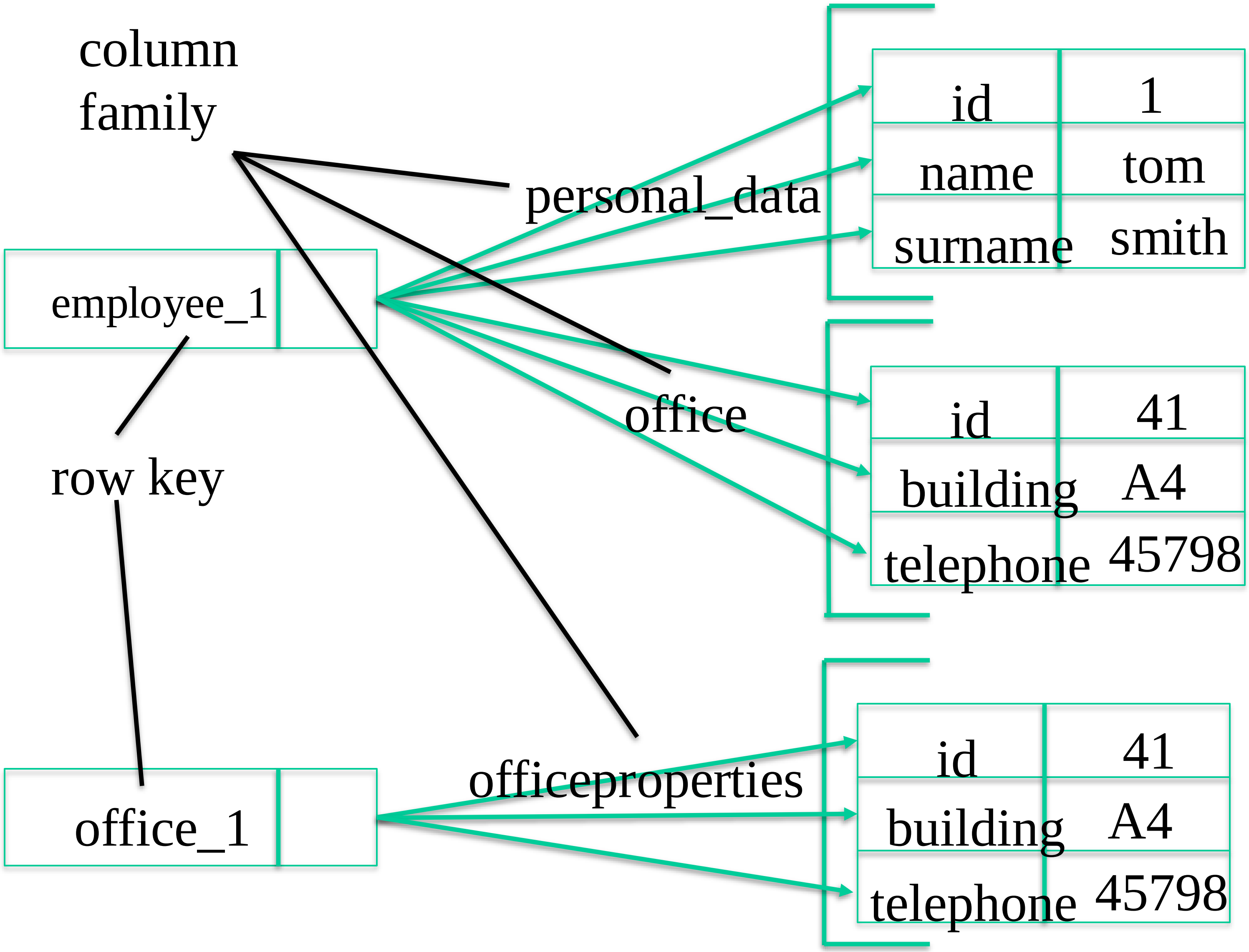
Key:"employee_1" →

```
{
  id:"1" .
  name:"Tom" .
  surname:"Smith" .
  office:{
        id:"41" .
        building:"A4" .
        telephone:"45798"
        }
}
```

Key:"office_1" →

```
{
  id:"41" .
  building:"A4" .
  telephone:"45798"
}
```

Column-oriented



NoSQL databases: the case of Graph-oriented databases

NoSQL databases: the case of Graph-oriented databases

We concentrate on graph-oriented databases, and consider two types of graph-oriented databases:

- Graph databases
- RDF databases

- Graph databases
- RDF databases

Introducing Graph databases

- A graph database is a database that uses graph structure with nodes, edges, and properties to represent and store data.

- A management systems for graph databases offers Create, Read, Update, and Delete (CRUD) methods to access and manipulate data.

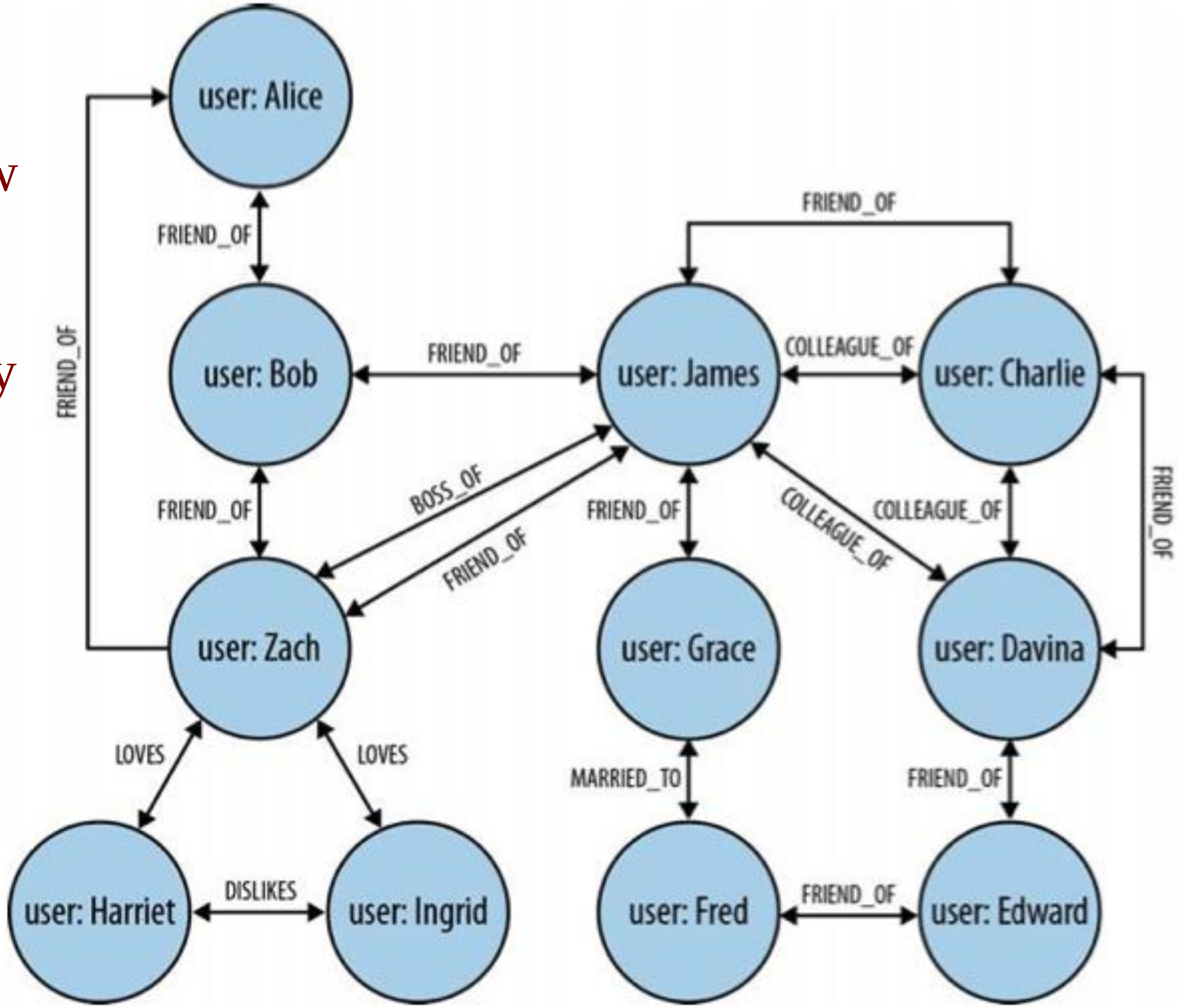
- Differently from other NoSQL management systems, Graph database systems (e.g., Neo4j) are generally optimized for *transactional performance*, and tend to guarantee ACID (transacton) properties.

Graph databases

- Graph databases are schemaless:
  - Thus they well behave in response to the dynamics of big data: you can accumulate data incrementally, without the need of a predefined, rigid shema
  - This does not mean that intensional aspects cannot be represented into a graph, but they are not pre-defined and are normally managed as data are managed (as, e.g., for RDF, discussed later on)
  - They provide flexibility in assigning different pieces of information with different properties, at any granularity
  - They are very good in managing sparse data
- Graph databases can be queried through declarative languages (some of them standardized): they can provide very good performances because essentially they avoid classical joins (*but performances depend on the kind of queries).*

Flexibility in graph databases

Incorporating new information is natural and simple: we simply introduce new nodes and/or edges



Graph Databases Embrace Relationships

- Obviously, graph databases are particulary suited to model situations in which the information is somehow "natively" in the form of a graph.

- The real world provides us with a lot of application domains: social networks, recommendation systems, geospatial applications, computer network, authorization and access control systems, to mention a few.

- The success key of graph databases in these contexts is the fact that they provide native means to represent objects (nodes) and relationships and links between objects (edges).

- Relational databases instead lack explicit relationships: they have to be simulated through the help of foreign keys, thus adding additional development and maintenance overhead, and "navigating" them require costly join operations.

Graph DBs vs Relational DBs- Example

Modeling friends and friends-of-friends in a relational database

Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Notice that in this example, PersonFriend is not simmetric: Bob is considered friend of Zach, but the converse does not necessarily hold.

Graph DBs vs Relational DBs- Example

Asking “which are the names of Alice’s friends?” (i.e., those that are considered friend of Alice) is easy

Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

```
SELECT p2.Person AS ALICE_FRIEND
FROM Person p1 JOIN PersonFriend pf ON
     p1.ID = pf.PersonID JOIN Person p2 ON
     pf.FriendID = p2.ID
WHERE p1.Person = 'Alice'
```

Graph DBs vs Relational DBs- Example

Things become more problematic when we ask, “which are the names of *Alice’s* friends-of-friends?”

Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

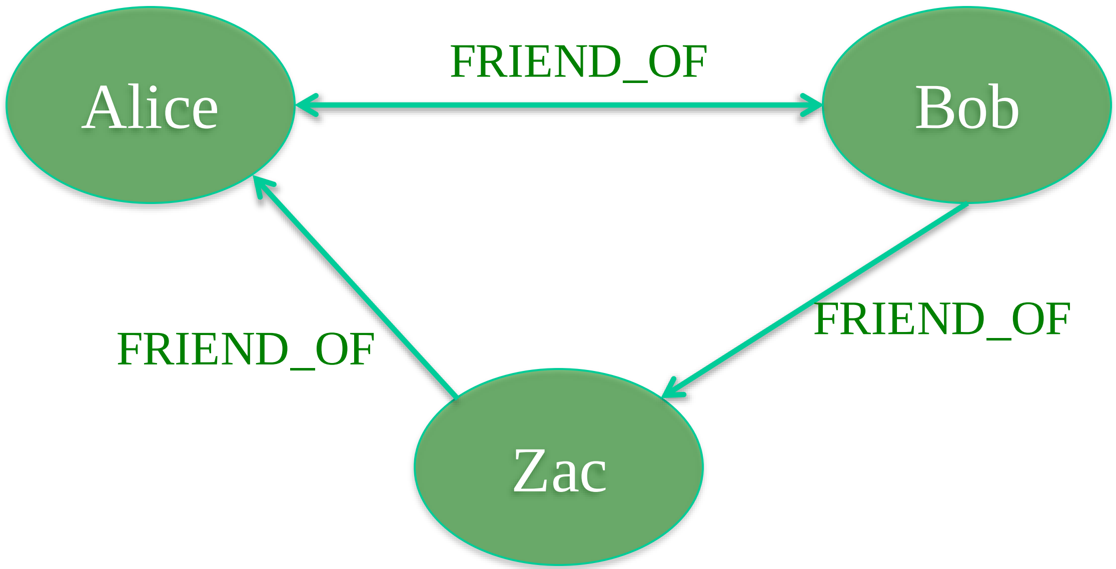
```
SELECT p2.Person AS ALICE_FRIEND_OF_FRIEND
FROM Person p1 JOIN PersonFriend pf1 ON
     p1.ID = pf1.PersonID JOIN PersonFriend pf2 ON
     pf1.FriendID = pf2.PersonID JOIN Person p2 ON
     pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

To exclude ‘Alice’ from her FOFs

Performances highly deteriorate when we go more in depth into the network of friends

Graph DBs vs Relational DBs- Example

Modeling friends and friends-of-friends in a graph database



Relationships in a graph naturally form paths. Querying means actually traversing the graph, i.e., following paths. Because of the fundamentally path-oriented nature of the data model, the majority of path-based graph database operations are extremely efficient.

Graph DBs vs Relational DBs- Experiment

The following table reports the results of an experiment aimed to find friends-of-friends in a social network, to a maximum depth of five, for a social network containing 1,000,000 people, each with approximately 50 friends.

Given any two persons randomly chosen, is there a path that connects them that is at most five relationships long?

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

From *Neo4j in Action. Jonas Partner, Aleksa Vukotic, and Nicki Watt. MEAP. 2012*

Graph DBs vs Relational DBs - Queries

SQL query

```
SELECT pf1.PersonID, pf2.FriendID
FROM PersonFriend pf1 JOIN PersonFriend pf2 ON
     pf1.FriendID = pf2.PersonID
UNION
SELECT pf1.PersonID, pf3.FriendID
FROM PersonFriend pf1 JOIN PersonFriend pf2 ON
     pf1.FriendID = pf2.PersonID JOIN PersonFriend pf3 ON
     pf2.FriendID = pf3.PersonID
UNION
....
```

Neo4J Cypher

```
MATCH (p:Person)-[:FRIEND_OF*2..5]->(fof:Person)
RETURN p, fof
```

Graph DBs vs Relational DBs- Queries*

- Relational Databases (querying is through joins)
  - The join operation forms a graph that is dynamically constructed as one table is linked to another table. The limitation is that this graph is not explicit in the relational structure, but instead must be inferred through a series of index-intensive operations.
  - Moreover, while only a particular subset of the data in the database may be desired (e.g., only Alice's friends-of-friends), all data in all queried tables must be considered in order to extract the desired subset.

- Graph Databases (querying is through traversal paths)
  - There is no explicit join operation because vertices maintain direct references to their adjacent edges. In many ways, the edges of the graph serve as explicit, "hard-wired" join structures (i.e., structures that are not computed at query time as in a relational database).
  - What makes this more efficient in a graph database is that traversing from one vertex to another is a constant time operation.

* From: Marko A. Rodriguez, Peter Neubauer: The Graph Traversal Pattern. Graph Data Management 2011: 29-46

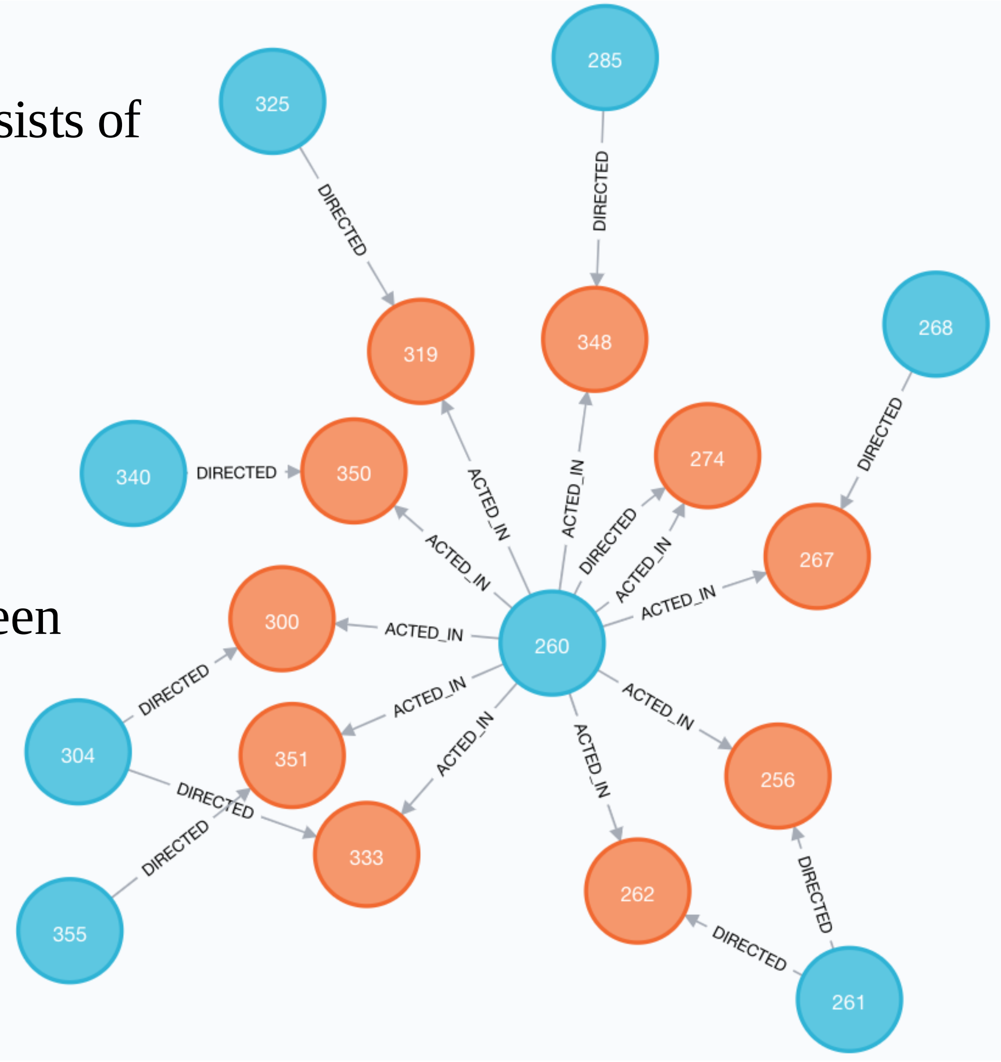
Abstract Data Type

- G = (V,E) over a finite alphabet Σ consists of
- V is a finite set of nodes or vertices,

  e.g. V={260, 274, 350, 351,…}
- Σ is a set of labels,

  e.g., Σ={DIRECTED, ACTED_IN}
- $E \subseteq V \times \Sigma \times V$ is a finite set of edges representing **binary** relationship between elements in V,

  e.g. E={(260, ACTED_IN, 350),
  (340, DIRECTED, 350),
  (260, DIRECTED, 274)…}



Basic Operations

Given a graph G, the following are operations over G:

- AddNode(G,x): adds node x to the graph G.
- DeleteNode(G,x): deletes the node x from graph G.
- Adjacent(G,x,y): tests if there is an edge from x to y.
- Neighbors(G,x): returns nodes y s.t. there is an edge from x to y.
- AdjacentEdges(G,x,y): returns the set of labels of edges from x to y.
- Add(G,x,y,l): adds an edge between x and y with label l.
- Delete(G,x,y,l): deletes the edge between x and y with label l.
- Reach(G,x,y): tests if there is a path from x to y.
- Path(G,x,y): returns a (shortest) path from x to y.
- 2-hop(G,x): returns the set of nodes y such that there is a path of length 2 from x to y, or from y to x.
- n-hop(G,x) : returns the set of nodes y such that there is a path of length n from x to y, or from y to x.

From [ABFRV13]

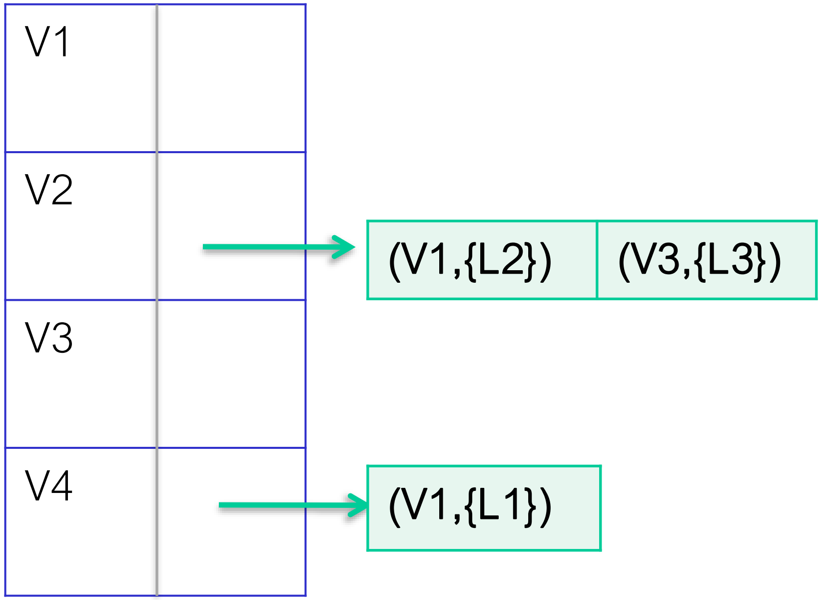
Implementation of Graphs

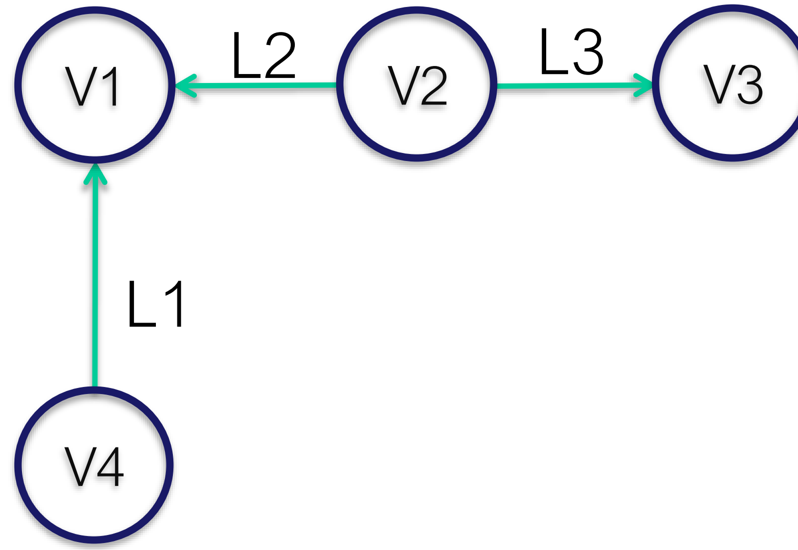
Adjacency List

- For each node I, a list of neighbors.
- If the graph is directed, adjacency list of i contains only the outgoing nodes of i.
- Cheaper for obtaining the neighbors of a node.
- Not suitable for checking if there is an edge between two nodes.

From [ABFRV13]

Adjacency List





- Adding a vertex just means adding it to the vertex set
- Adding an edge means adding the end-point of it to the starting vertex's neighbour set
- It is easy to go from a vertex to its neighbours, because the vertex stores them all
- Testing for adjacency means searching for the second vertex within the neighbours of the first vertex
- Getting all edges is more difficult, because edges don't exist as objects. You need to iterate over the neighbours of each vertex in turn, and construct the edge from the vertex and the neighbour

Adapted From [ABFRV13]

Implementation of Graphs

Adjacency List

- For each node a list of neighbors.
- If the graph is directed, adjacency list of i contains only the outgoing nodes of i.
- Cheaper for obtaining the neighbors of a node.
- Not suitable for checking if there is an edge between two nodes.

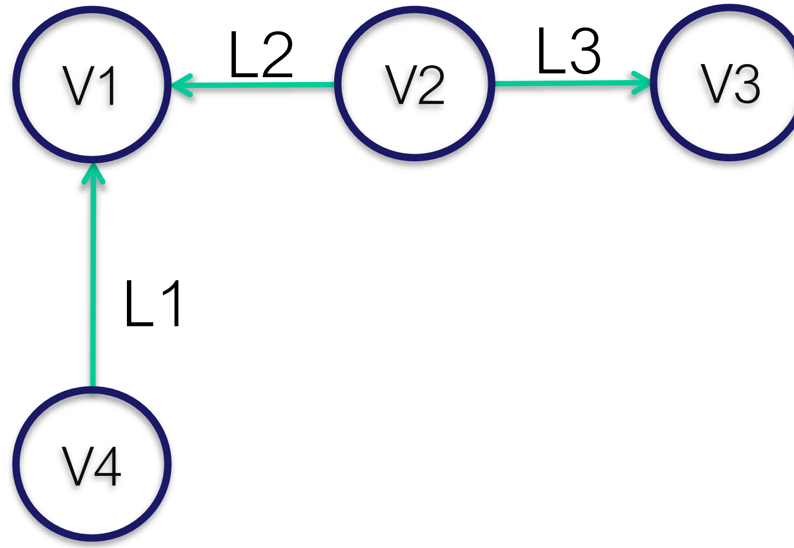
Adjacency Matrix

- Bidimensional graph representation.
- Rows represent source vertices.
- Columns represent destination vertices.
- Each non-null entry represents that there is an edge from the source node to the destination node.

From [ABFRV13]

Adjacency Matrix

	V1	V2	V3	V4
V1				
V2	{L2}		{L3}	
V3				
V4	{L1}			



- Memory used is a constant $|V|^2$
- Adding a vertex: add a row and column to the matrix. Removing a vertex: remove its row and column. The need of these operations makes the adjacency matrix unsuitable for graphs in which vertices are frequently added and removed. Adding and removing edges is easy however.
- To get neighbours, look along the vertex's row
- To determine adjacency, look for a non-null value at the intersection of the first vertex's row and the second vertex's column
- The matrix can be sparse

Adapted From [ABFRV13]

Implementation of Graphs

Adjacency List

- For each node a list of neighbors.
- If the graph is directed, adjacency list of i contains only the outgoing nodes of i.
- Cheaper for obtaining the neighbors of a node.
- Not suitable for checking if there is an edge between two nodes.

Adjacency Matrix

- Bidimensional graph representation.
- Rows represent source vertices.
- Columns represent destination vertices.
- Each non-null entry represents that there is an edge from the source node to the destination node.

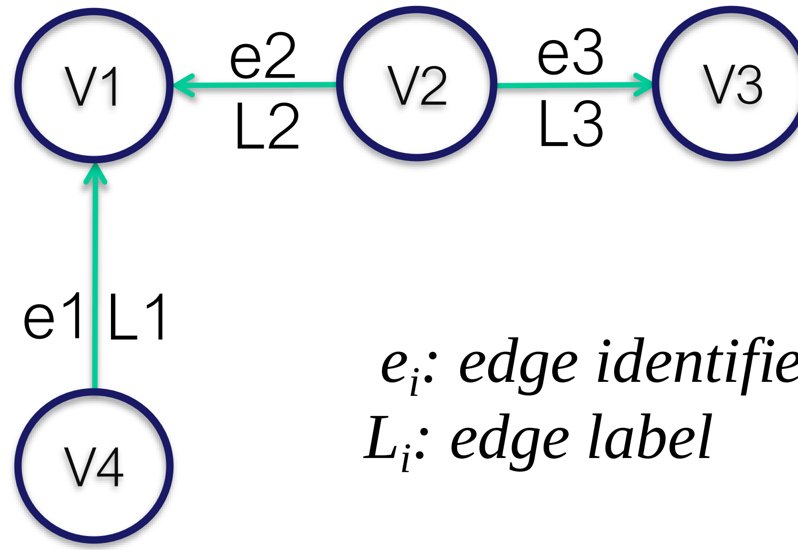
Incidence Matrix

- Bidimensional graph representation.
- Rows represent Vertices.
- Columns represent edges
- A non-null entry represents that the source vertex is incident to the Edge.

From [ABFRV13]

Incidence Matrix

	e1	e2	e3
V1	destination{ L1}	destination {L2}	
V2		source {L2}	source {L3}
V3			destination {L3}
V4	source {L1}		



e_i: edge identifier
L_i: edge label

Properties:

- Memory usage is O(|V|x|E|)
- Adjacent(G,x,y): scan the row of x, and for each L such that (x,L) is labeled 'source', you have a constant time access to the cell (y,L)
- Neighbors(G,x): you have to scan the entire matrix
- AdjacentEdges(G,x,y): similar to Adjacent(G,x,y)
- Adding/removing a vertex: add/remove a row to the matrix. If the removed vertex is the source/destination of an edge *L,* remove the *L* column
- Adding/removing an edge: add/remove a column to the matrix.

Adapted From [ABFRV13]

Traversal Search

Breadth First Search

- Expands shallowest unexpanded nodes first.
- Unexpanded nodes are stored in queue.

Depth First Search

- Expands deepest unexpanded nodes first.
- Unexpanded nodes are stored in a stack.

From [ABFRV13]

Breadth First Search

Notation:

1 Starting Node

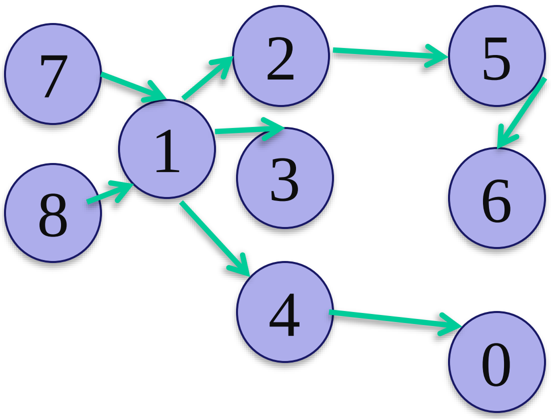
First Level Visited Nodes

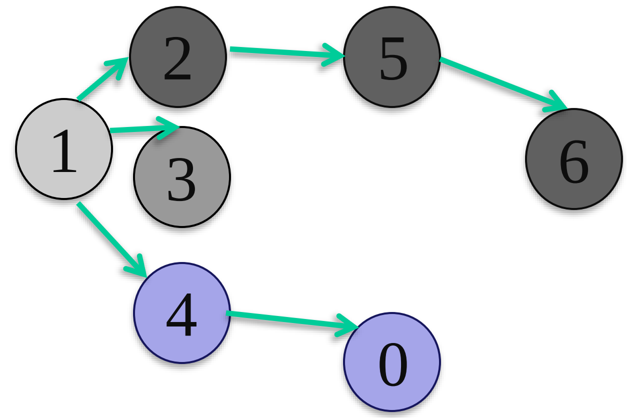
Second Level Visited Nodes

Third Level Visited Nodes

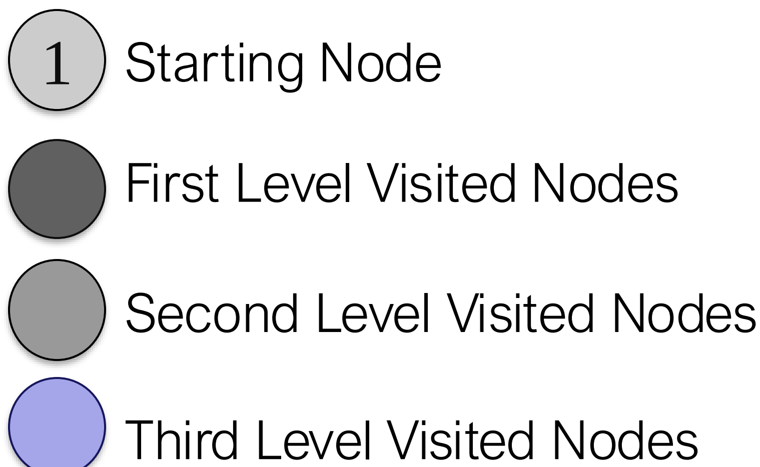
From [ABFRV13]

Depth First Search





Notation:



From [ABFRV13]

Querying Graph DBs

- A traversal refers to visiting elements (i.e. vertices and edges) in a graph in some algorithmic fashion. Query languages for graph databases allow for recursively traversing the labeled edges while checking for the existence of a path whose label satisfies a particular regular condition (i.e., expressed in a regular language).

- Basically, a *graph database* $G = (V,E)$ over a finite alphabet Σ consists of a finite set V of nodes and a set of labeled edges $E \subseteq V \times \Sigma \times V$.

- A path π in G from node v_0 to node v_m is a sequence of the form

$$(v_0, a_1, v_1)(v_1, a_2, v_2) \ldots (v_{m-1}, a_m, v_m)$$

where (v_{i-1}, a_i, v_i) is an edge in E, for each $1 \leq i \leq m$. The *label* of π, denoted $\lambda(\pi)$, is the string $a_1a_2...a_m \in \Sigma^*$.

Regular Expressions: syntax

Syntax of regular expressions over the alphabet Σ:

`L ::= s | L·L | L|L | L* | L+ | L? | (L)`

where, intuitively,
- `s` is an element of the alphabet Σ
- `·` denotes string concatenation (it can be omitted, i.e., `LL=L·L`),
- | denotes an OR, i.e., `L1|L2` in an expression matching with `L1` or `L2`
- * is the Kleen operator, denoting concatenation of 0 or any number of string matching the expression `L`
- + is similar to * but there must be at least one occurrence of a string mathing the expression `L`
- `?` denotes 0 or 1 occurrences of the string matching the `L` expression.

Examples:
- Has ancestors:
  `isChildOf+`
- Are cousins (for simplicity, an individual can be cousin of hersef):
  `isChildOf·isChildOf·hasChild·hasChild`

Regular Expressions: semantics

The semantics of regular expressions tells us which are the strings that belong to the language defined by each regular expressions. To define the semantics of Regular Expressions, we proceed recursively on the basis of the form of the expression:

- *lang*(s) = { s }
- *lang*(`L` · `L`) = { s1 · s2 | s1 ∈ *lang*(s1) and s2 *lang*(s2) }
- *lang*(`L` | `L`) = {s ∈ *lang*(s1) } ∪ {s ∈ *lang*(s2) }
- *lang*(`L*`) = { s' | s' is a sequence of symbols in `L` of length ≥ 0 }
- *lang*(`L+`) = { s' | s' is a sequence of symbols in `L` of length ≥ 1 }
- *lang*(`L?`) = { the empty string } ∪ `L`
- *lang*((`L`))= *lang*(`L`)

Examples

Σ = { a,b,c,d }, and assume **ε** denotes the empty string

- regular expression L = ab*
  lang(L) = { a, ab, abb, abbb, abbbb, abbbbb, abbbbbb, ... }

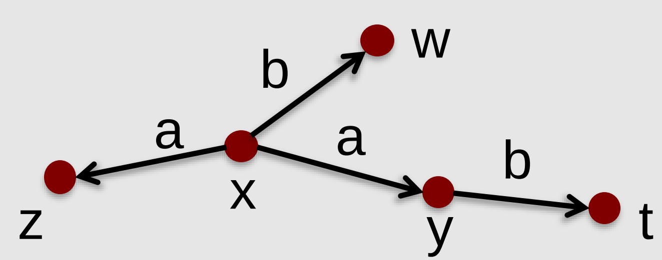
- regular expression L = (a | (b · c)$^+$)?
lang(L) = {**ε**, a, bc, bcbc, bcbcbc, bcbcbcbc, bcbcbcbcbc, ... }

Regular Path Queries

Let G = (V,E) be a *graph database* over a finite alphabet Σ of labels.
A Regular path query for G is a regular expression L over Σ. The evaluation

L(G) of L over G is the set of pairs (u,v) of nodes in V for which there is a path π in G from u to v such that the string λ(π) is in the language L.

Example:



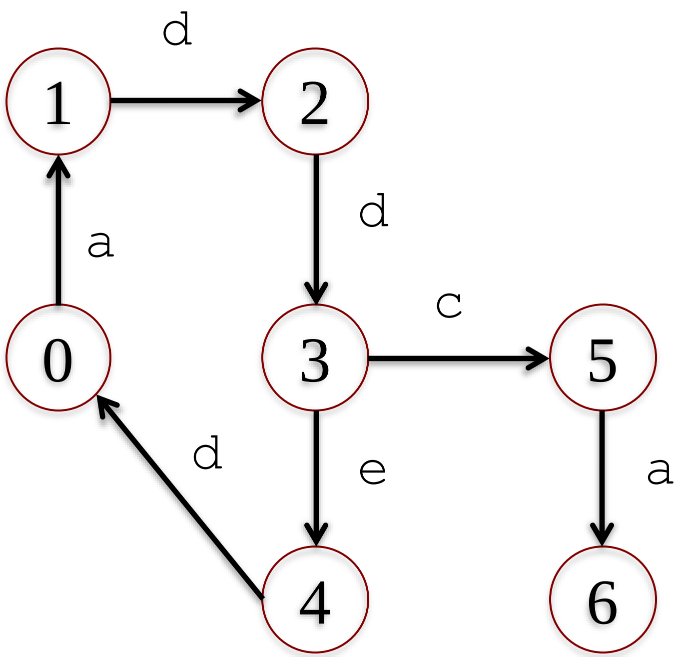
Graph G (with Σ={a,b})

regular expression L = ab*

L(G) = {(x,y), (x,z), (x,t)}

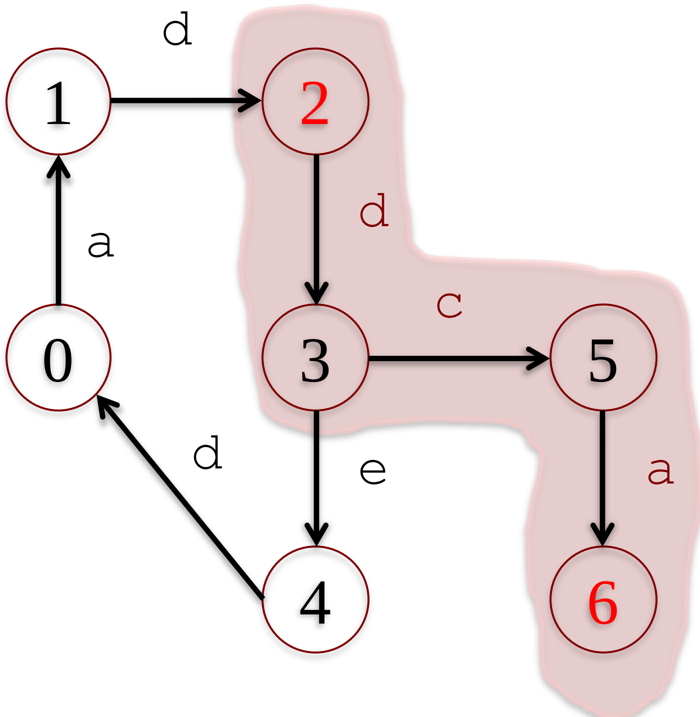
Query languages for graph databases typically extend this class of queries

Example



regular expression: `d+(c|e)a`

Example

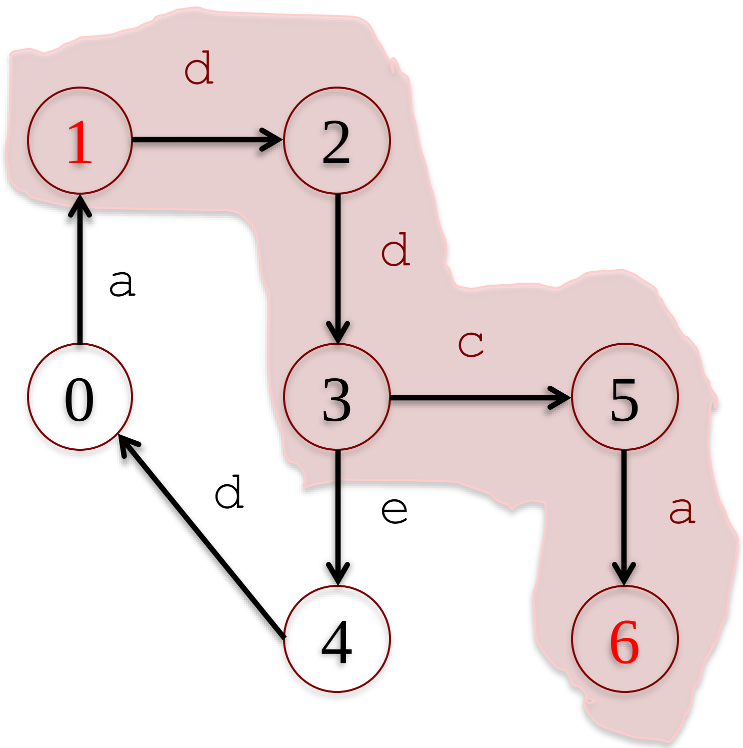


regular expression: `d+(c|e)a`

matching path: `dca`

node pair: (2,6)

Example



regular expression: `d+(c|e)a`

matching path: `ddca`

node pair: (1,6)

Exercises

Write the regular expressions over the alphabet

`{isFriendOf, isChildOf, hasChild}`

that in a regular path query allow to retrieve pairs of nodes (b,c) such that:

1) b is friend of c, or a parent of b is friend of a parent of c;
2) b is a nephew of c;
3) b is friend of c, or c is a child of a friend of b.

Note: assume that in the graph isChildOf is the inverse of hasChild, i.e., given a graph G, if (x isChildOf y) is an edge of G, then (y hasChild x) is an edge in G, and vice-versa.

Graph Database Management Systems

A Graph Database Management System (GDBMS) is a system that manages graph databases. Some GDBMSs are:

Native graph storage and processing

- Some GDBMSs use *native graph storage*, which is optimized and designed for storing and managing graphs.
- In contrast to relational DBMSs, these GDBMSs do not store data in disparate tables. Instead they manage a single data structure (e.g., adjacency lists, adjacency matrices, incidence matrices).
- Coherently, they adopts a *native graph processing*: they leverage index-free adjacency, meaning that connected nodes physically “point” to each other in the database.

Index-free adjacency

- A database engine that utilizes index-free adjacency is one in which each node maintains direct references to its adjacent nodes; each node therefore acts as a micro-index of other nearby nodes, which is much cheaper than using global indexes.

- In other terms, a (graph) database *G* satisfies the index-free adjacency if the existence of an edge between two nodes v_1 and v_2 in *G* can be tested on those nodes and does not require to access an external, global, index.

- Locally, each node can manage a specific index to speed up access to its outgoing edges.

Non-native graph storage

- Not all graph database systems use native graph storage, however. Some serialize the graph data into a relational database, object-oriented databases, or other types of general-purpose data stores.
- GDBMSs of this kind do not adopt index-free-adjacency, but resort to classical relational index mechanisms.

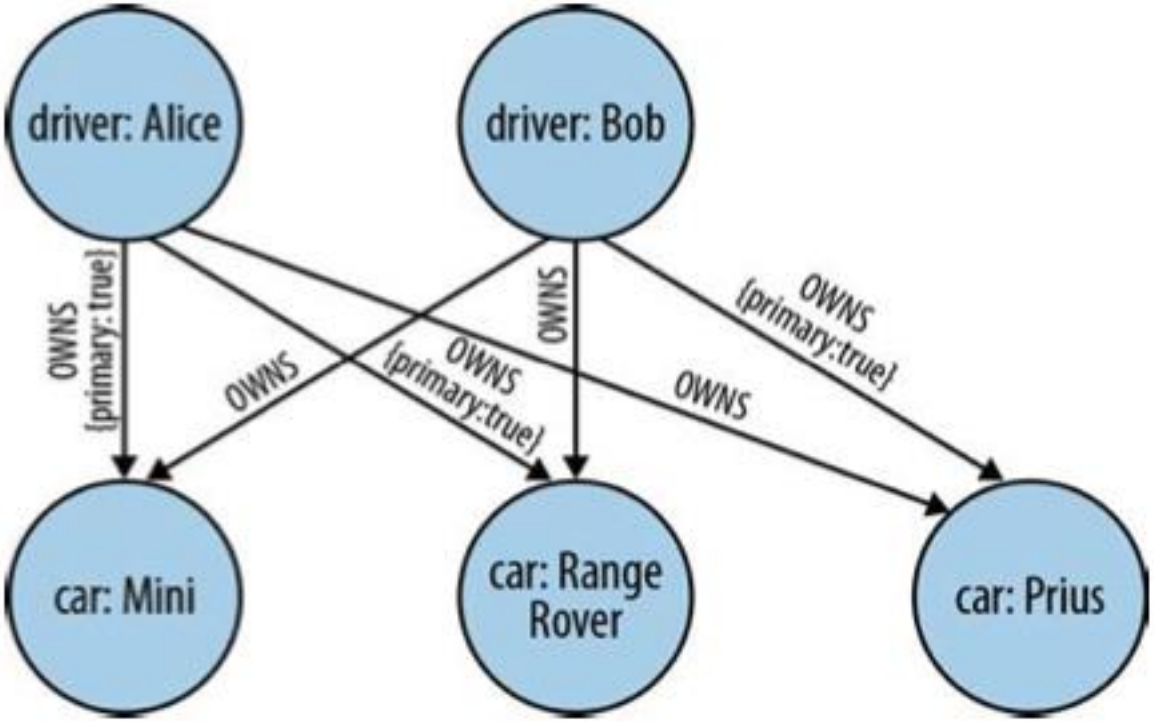
Note: Some authors consider index-free-adjacency a distinguishing property for graph databases (i.e., a DBMS not using index-free-adjacency is not a Graph DBMS). Alternatively (as we do in these slides) it is possible to classify as graph database any database that from the user's perspective *behaves* like a graph database (i.e., exposes a graph data model through CRUD operations)

Types of graph databases

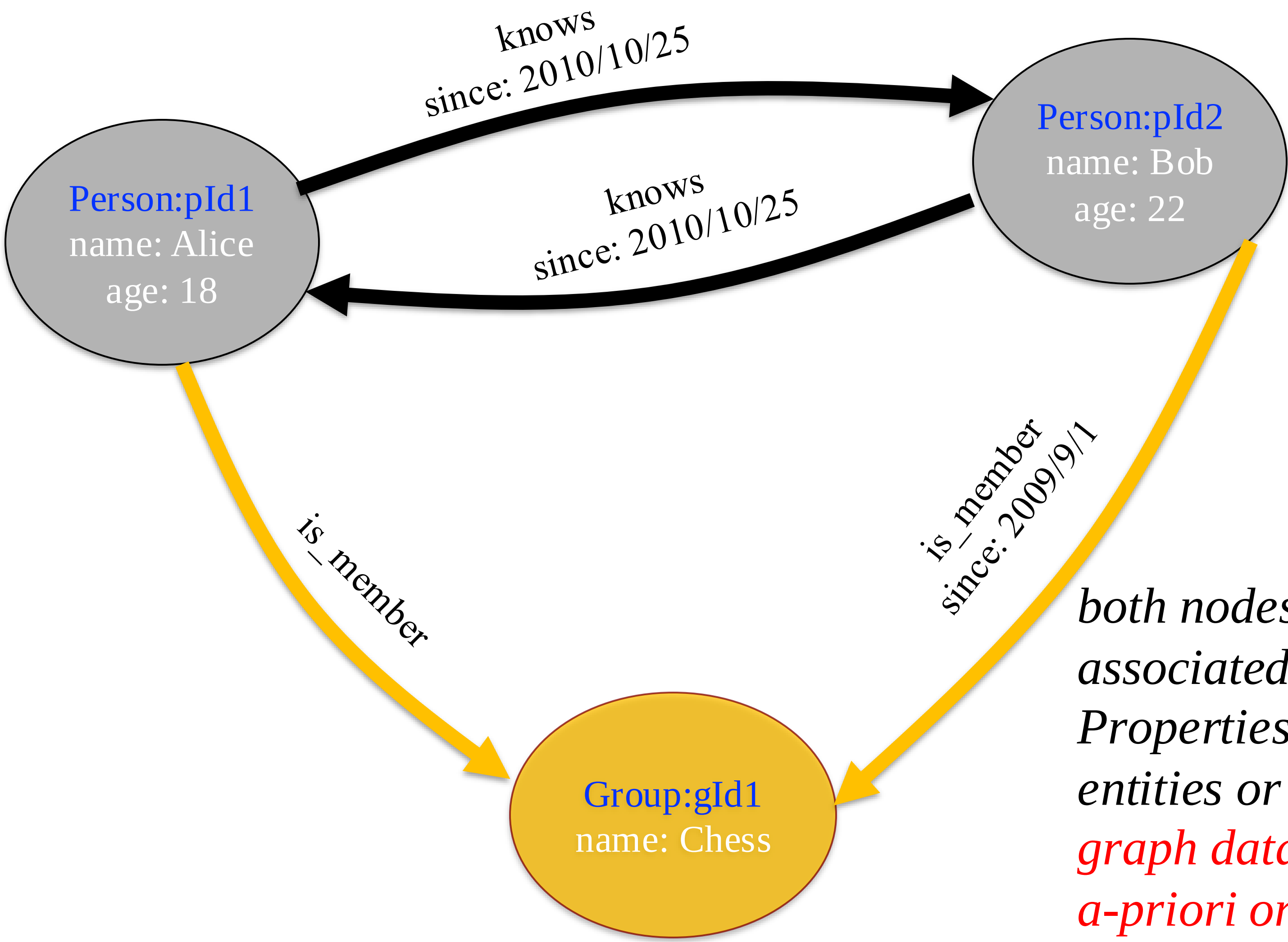
- Remember that Graph-oriented databases are classified into:
  - Graph databases
  - RDF databases
- There are several different data models for graph-oriented databases, which somehow generalizes the basic definition we have seen before, including
  - property graphs,
  - hypergraphs,
  - triple stores.

Property-graph databases

- A property graph is a labeled directed multigraph $G = (V, E)$ over an alphabet Σ of labels where every node $v \in V$ and every edge $e \in E$ can be associated with a set of pairs <*attribute, value*>, called properties.
- Each edge represents a link between nodes and is associated with a label, which is the name of the (intensional) relationship which the link is instance of.
- Nodes are usually classified according to a "type" (e.g., driver, car)



Property-graph databases

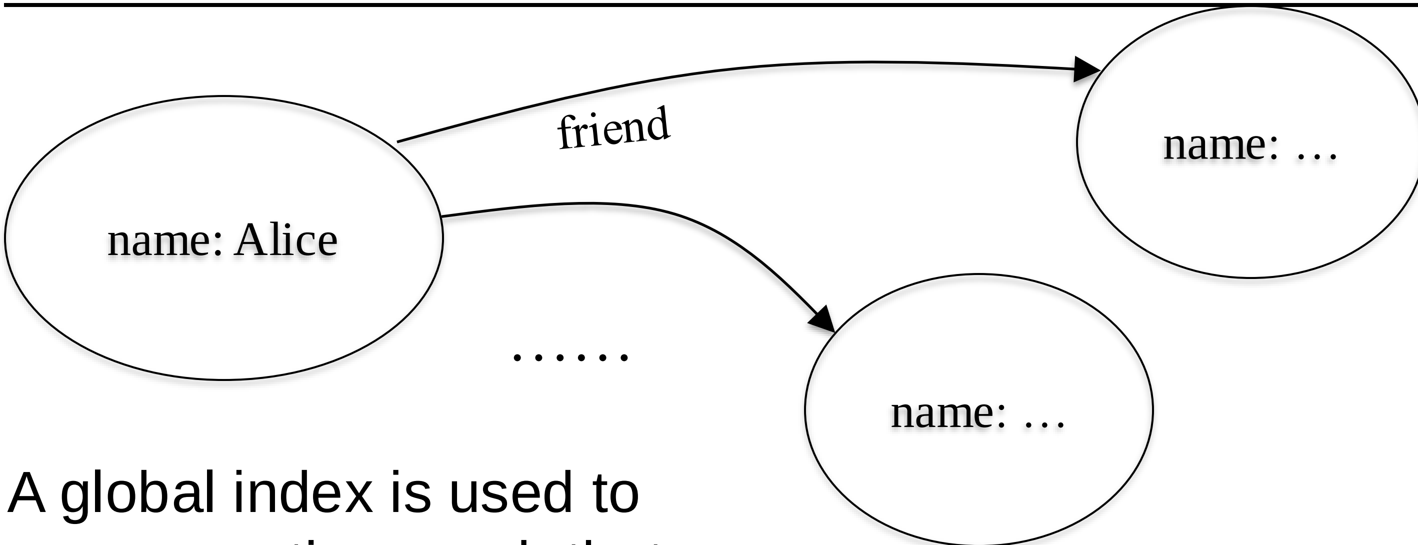


both nodes and edges can be associated to properties. Properties act as attributes of entities or relationships, but in graph databases there is no a-priori or rigid schema

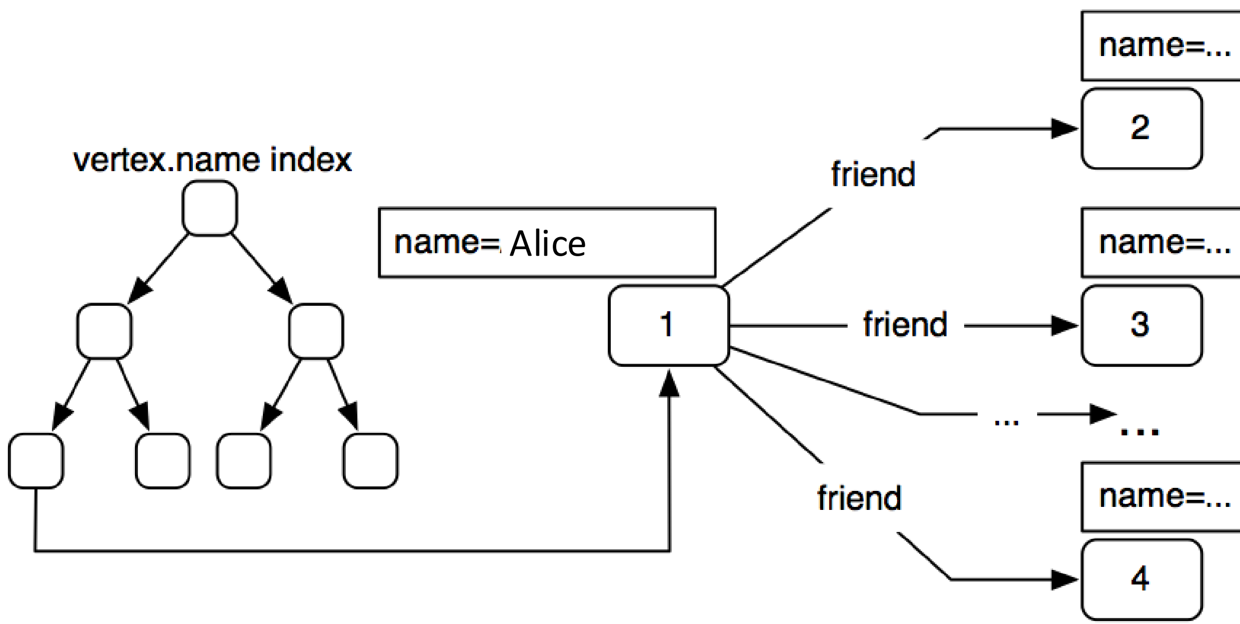
Querying property-graph databases

- As we have seen, basic query languages for graph databases, as Regular Path Queries (RPQs), essentially only retrieve their topology.
- However, in property-graph databases we also want to access data stored at the nodes and the edges (i.e., the properties).
- RPQs do not allow for this, but tailored languages (as the Neo4J Cypher) exist that enable property retrieval
- The execution of queries that access properties, however, besides exploiting adjacency, somehow relies on relational mechanisms:
  - In the property graph model, it is common for the properties of vertices (and sometimes edges) to be globally indexed using a tree structure analogous, in many ways, to those used by relational databases.

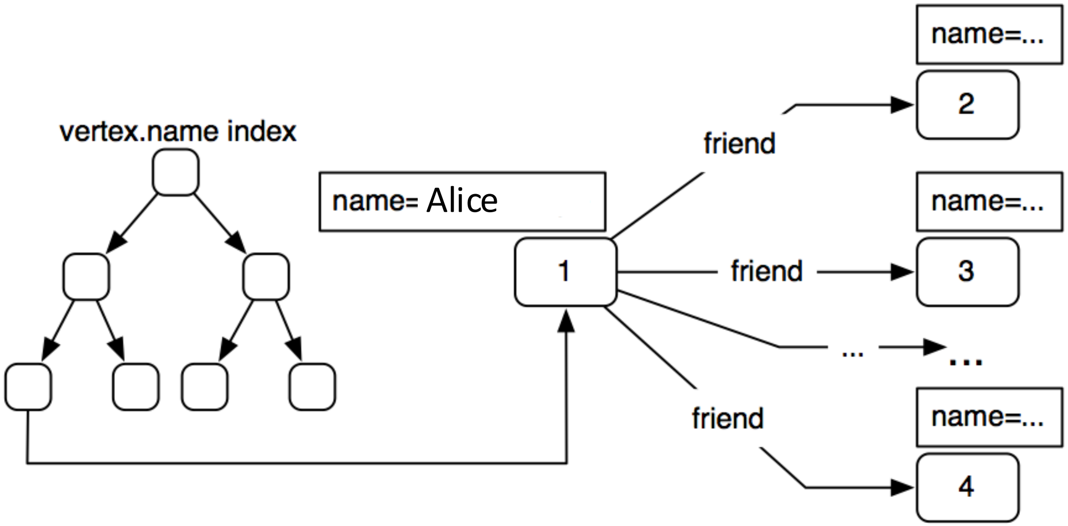
People and their friends example



A global index is used to access vertices such that name= "Alice"



People and their friends example



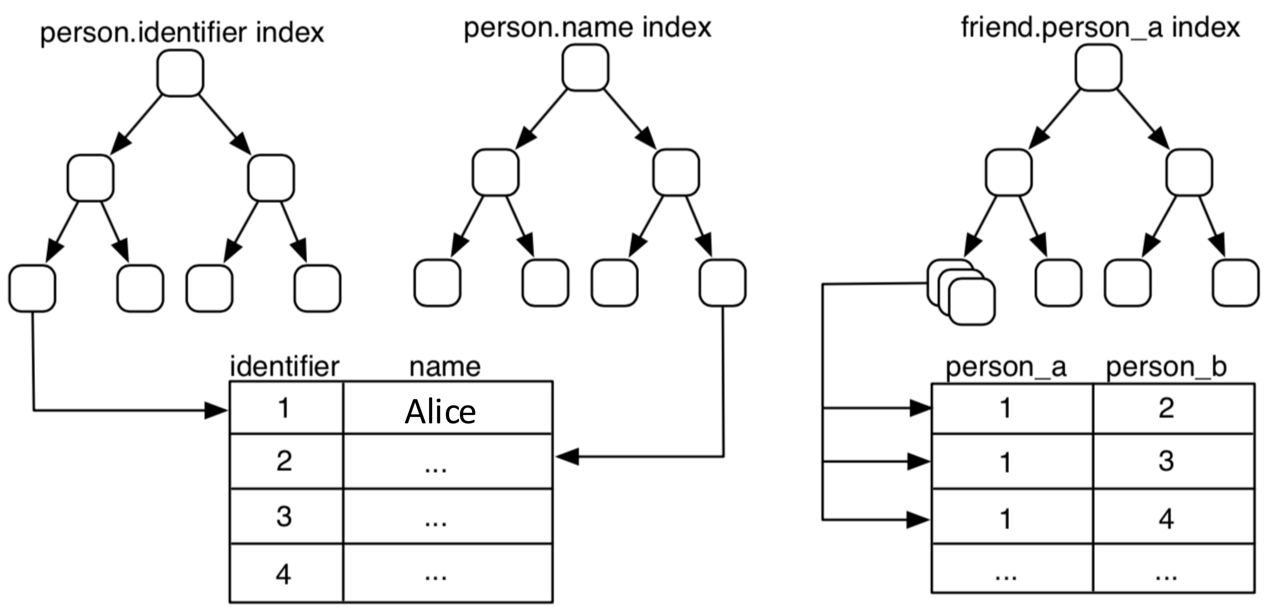
1. Query the vertex.name index to find all the vertices with the name “Alice” [O($\log_2 n$)] (where n is the number of nodes with the property “ name”)
2. Given the vertex returned, get the k friend edges starting from this vertex. [O(k + x)]] (where k is the number of friend edges and x is the number of the outgoing edges with a label different from friend)
3. Given the k friend edges retrieved, get the k vertices on the heads of those edges [O(k)] (getting one vertice is constant, thus getting k vertices costs O(k))
4. Given these k vertices, get the k name properties of these vertices. [O(k*y)] (where y is the number of properties in each vertex)

Same query in a relational store

Assume there are two tables: person(id, name) and friend(pers_a, pers_b).

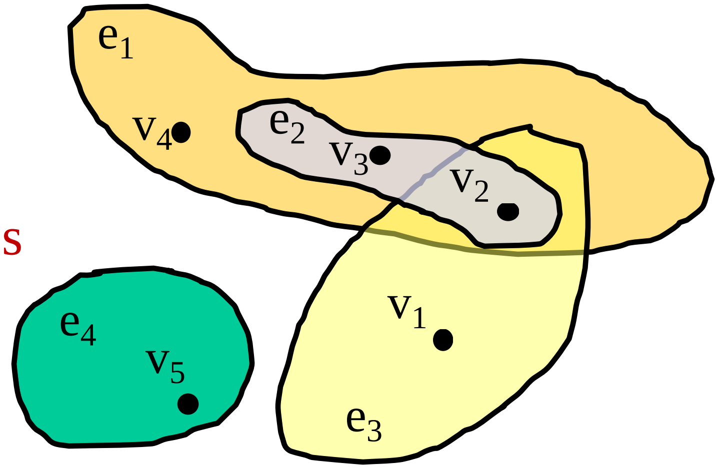
Suppose the problem of determining the name of all of Alice's friends.

1. Query the person.name index to find the row in person with the name "Alice" [$O(\log_2 n)$] (n is the number of rows in person)
2. Given the person row returned by the index, get the identifier for that row [$O(1)$]
3. Query the friend.person_a index to find all the rows in friend with the identifier from the previous step [$O(\log_2 z)$] (z is the number of nodes in the index friend.person_a; z <m, where m is the total number of rows in friend)
4. Given each of the k rows returned, get the person_b identifier for those rows [$O(k)$]
5. For each of the k friend identifiers, query the person.identifier index [$O(k \log_2 n)$]
6. Given the k person rows, get the name value for those rows. [$O(k)$]

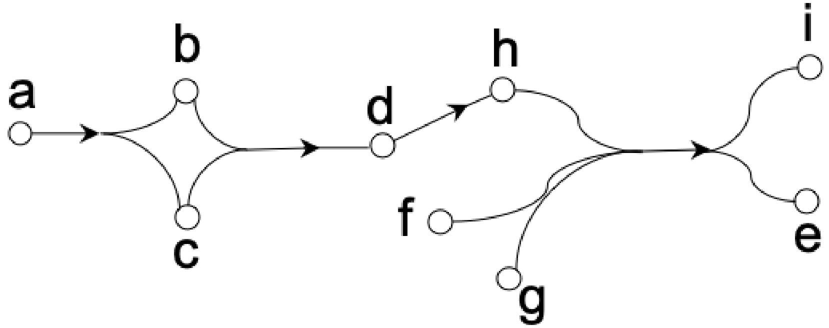


Hyper-graph databases

- Hypergraphs generalize a graph since they allow an edge to connect any number of vertices
- Formally, an hypergraph is a pair H = (V,E) where V is a finite set of vertices, and E is a set of non-empty subsets of V called hyperedges

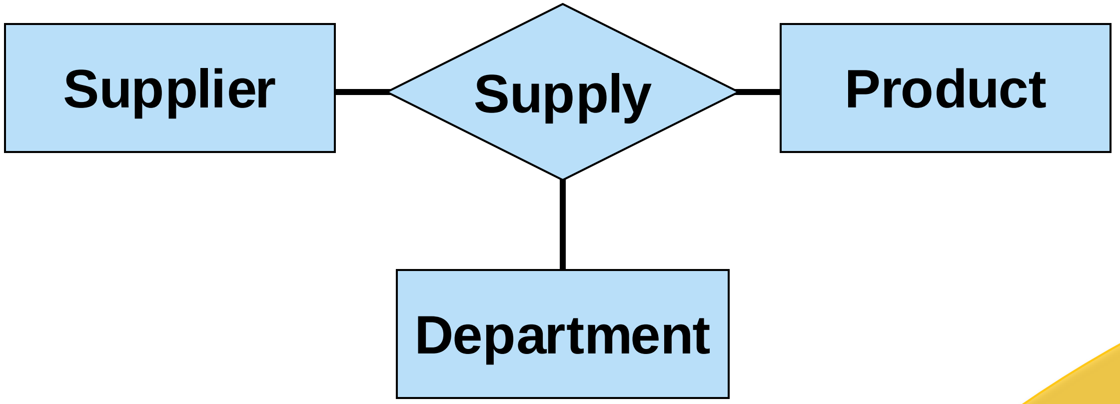


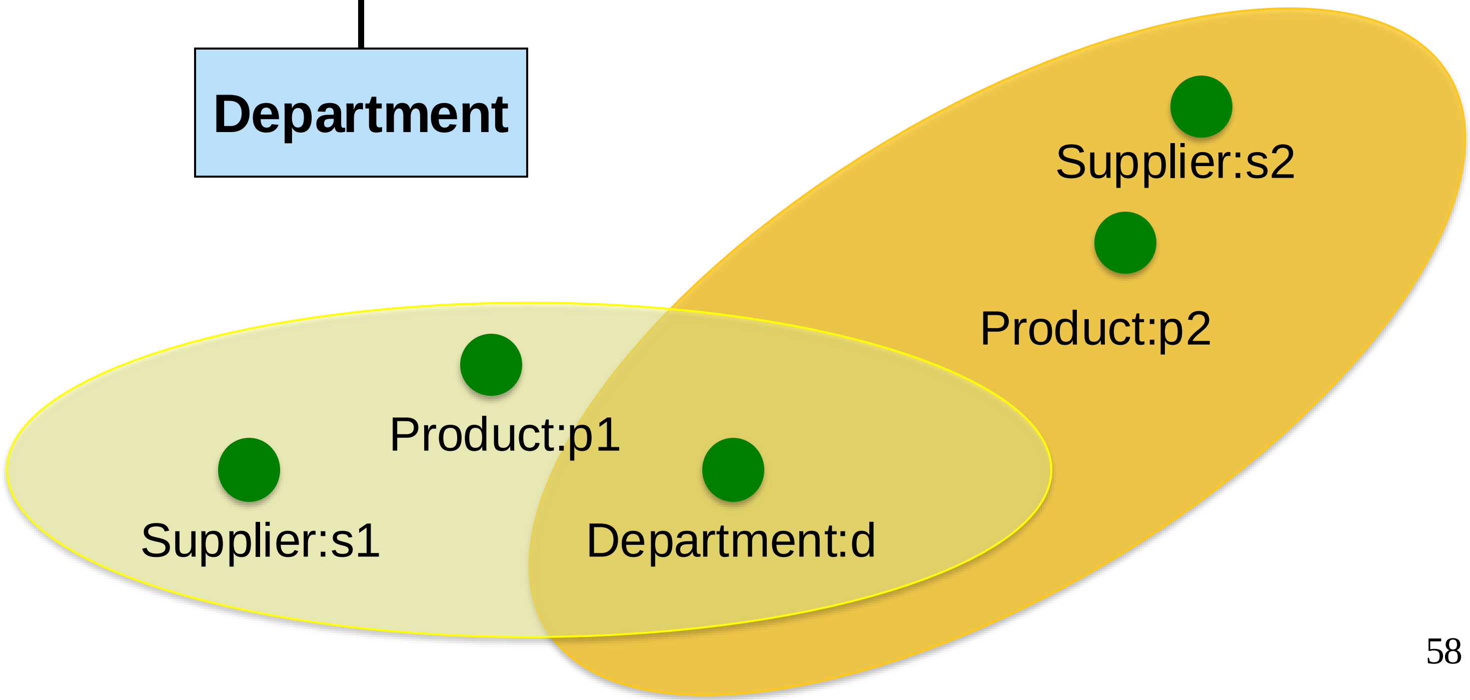
- Given a finite set of vertices V, a directed hyperedge over V is an ordered pair (T,H) of disjoint subsets of V. T is called the tail and H is called the head of the hyperedge. A directed hypergraph is a pair (V,A), where V is a finite set of vertices and A is a set of directed hyperhedges above V.



Hyper-graph databases

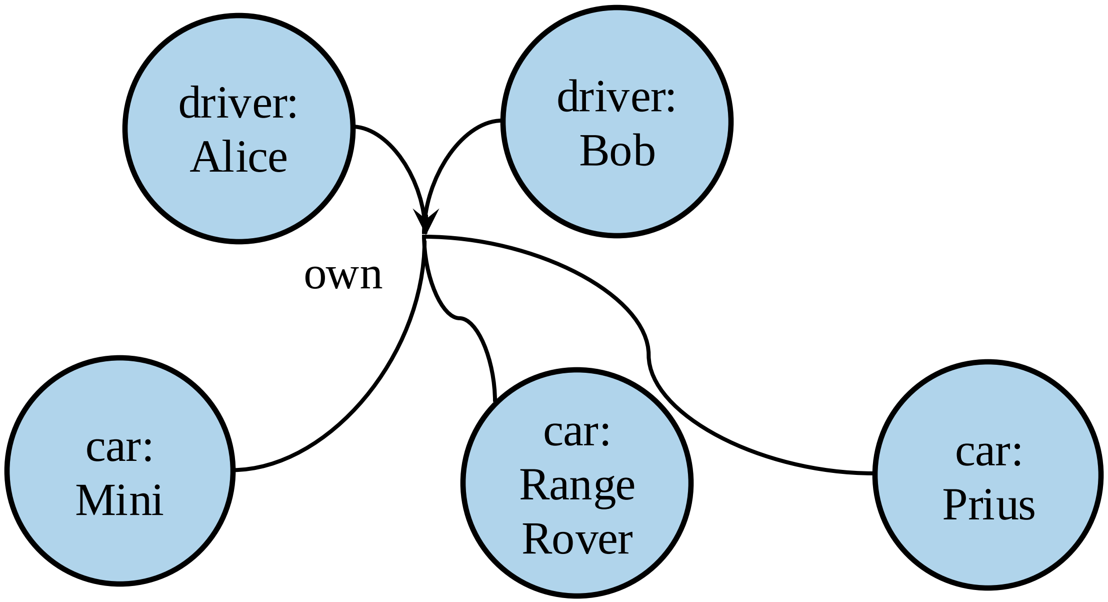
- A relationship (called a hyper-edge) can connect any number of nodes, thus can be useful where the domain consists mainly of **n-ary relationships**





Hyper-graph databases

- In the example below we can represent with a unique hyper-edge that Alice and Bob own together a Mini, a Range Rover and a Prius car. However, we loose some flexibility in specifying some properties (e.g., who is the primary owner)
- In this example the hyper-graph is oriented.



- Notice that any hypergraph database can be encoded into a graph database

RDF triple stores (see later)

- Triple stores come from the Semantic Web movement, where researchers are interested in large-scale knowledge inference by adding semantic markup to the links that connect web resources.
- A triple is a *subject-predicate-object* data structure. Using triples, we can capture facts, such as “Ginger dances with Fred” and “Fred likes ice cream.”
- The standard way to represent triples and query them is by means of RDF and SPARQL, respectively.

Note: triple stores turned out to be a particularly useful format to exchange information on the Web and have become nowadays very popular, especially in the Semantic Web context. Here however we are more interested in data management aspects rather that in the semantic characteristics of RDF and SPARQL.

Bibliography

- Some examples and figures on graph databases are taken from: I. Robinson, J. Webber, & E. Eifrem. Graph Databases. O'Reilly. 2013. Available at http://graphdatabases.com/
- The slides from 18 to 27 are taken from: Maribel Acosta, Cosmin Basca, Alejandro Flores, Edna Ruckhaus, Maria-Esther Vidal. Semantic Data Management in Graph Databases. ESWC-13 tutorial ([ABFRV13]), with minor adaptations.
- Additional Bibliography:
  - Marko A. Rodriguez, Peter Neubauer: The Graph Traversal Pattern. CoRR abs/1004.1001 (2010). Available at https://arxiv.org/abs/1004.1001
  - Andreas Schmidt, Iztok Savnik. Overview of Regular Path Queries in Graphs. DBKDA (2015). Available at https://www.iaria.org/conferences2015/filesDBKDA15/graphsm_overview_of_regular_path_queries_in_graphs.pdf

- In order to conclude the part on «Graph Databases», we will have a presentation of the NEO4J graph database management system. See the slides on that part!

- After that, we will study RDF databases.

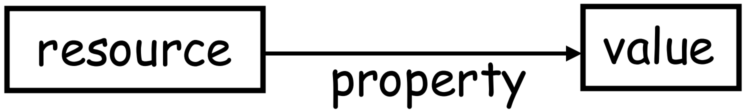
- Graph databases
- RDF databases

Resource Description Framework (RDF)

- RDF is a data model
  - ✓ the model is domain and application neutral
  - ✓ besides viewing it as a graph data model, it can be also viewed as an object-oriented model (object/attribute/value)
- RDF database = set of RDF triples that can be seen as a graph
- triple = expression (statement)

(subject, predicate, object)

- subject = resource
- predicate = property (of the resource)
- object = value (of the property)



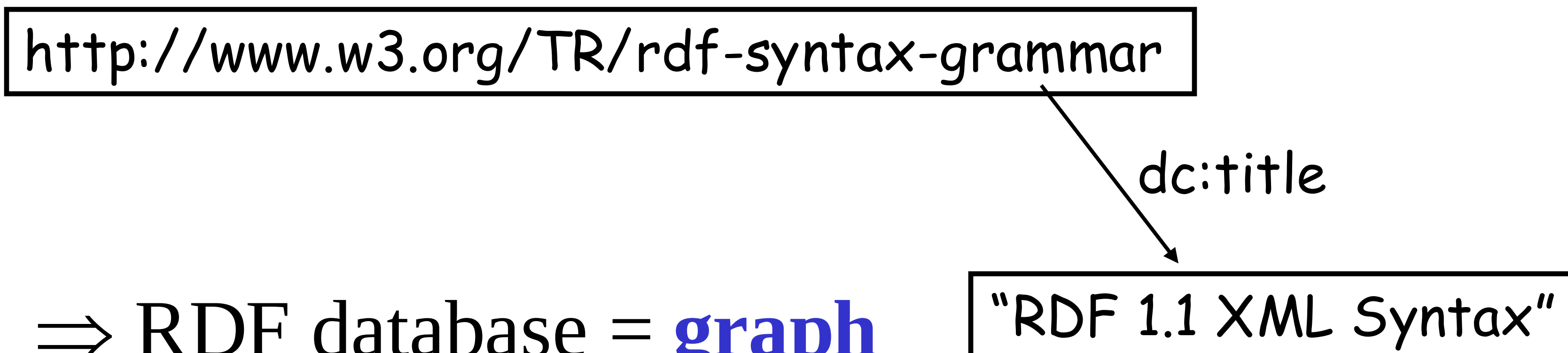
RDF triple

Example:

the document at http://www.w3.org/TR/rdf-syntax-grammar has "RDF 1.1 XML Syntax" as title

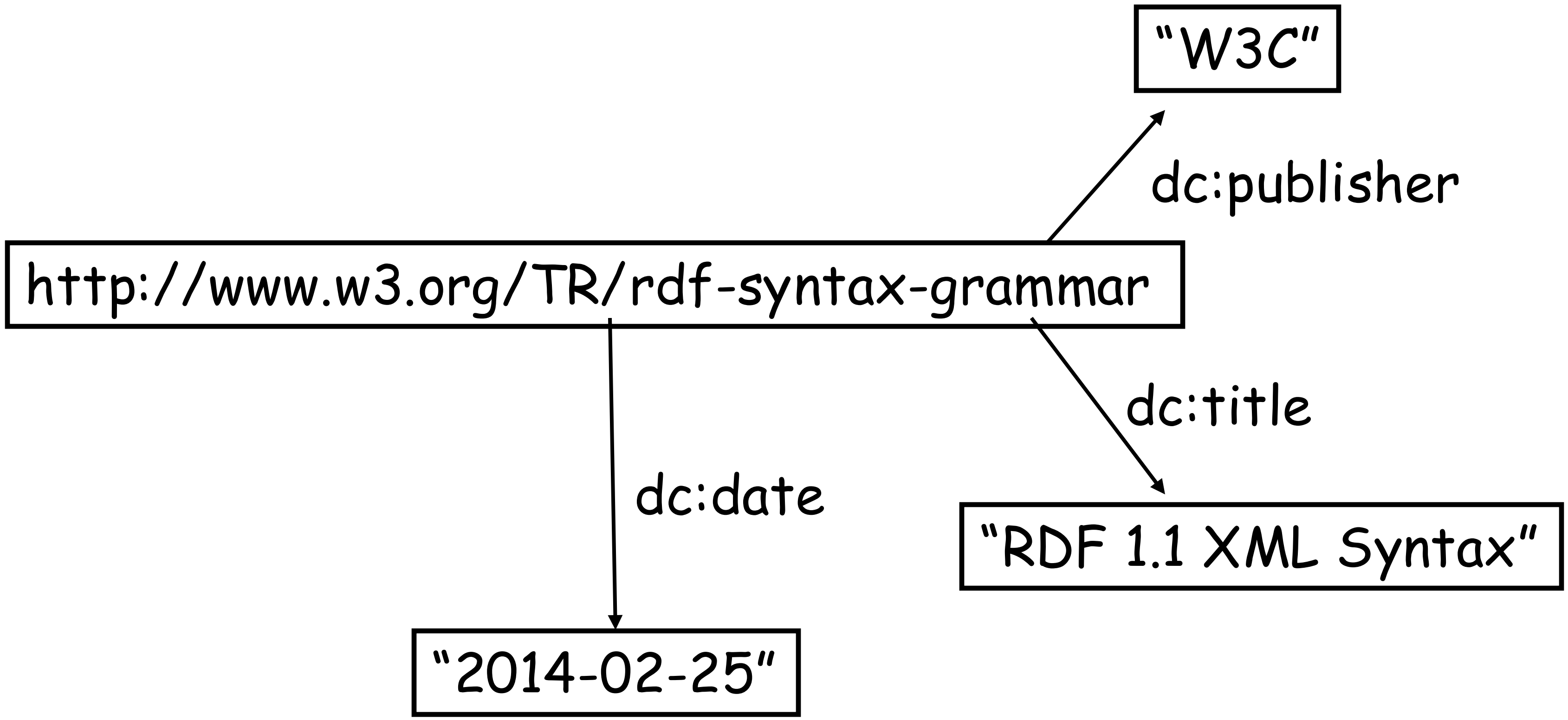
with a triple:

http://www.w3.org/TR/rdf-syntax-grammar   dc:title   "RDF 1.1 XML Syntax"



$\Rightarrow$ RDF database = **graph**

RDF graph: example



Node and edge labels in RDF graphs

node and edge labels:

- **URI -** Uniform Resource Identifier
- **Literal**, string that denotes a fixed resource (i.e., a value)
- **blank node**, i.e., an anonymous label, representing unnamed resources

but:

- a literal can only appear in object positions (that is, literals are end nodes in an RDF graph)
- a blank node can only appear in subject or object positions
- URIs can be used as subjects, objects, and predicates
- The same URI can be used both as predicate and as subject/object, i.e., graph nodes can be used as edge labels.

URI

- A **Uniform Resource Identifier** (**URI**) is a unique sequence of characters that identifies a logical or physical resource used by web technologies
- Some URIs provide a means of locating and retrieving information resources on a network (either on the Internet or on another private network, such as a computer filesystem or an Intranet), these are Uniform Resource Locators (URLs). Other URIs provide only a unique name, without a means of locating or retrieving the resource or information about it, these are Uniform Resource Names (URNs).
- For example, in the International Standard Book Number (ISBN) system, *ISBN 0-486-27557-4* identifies a specific edition of Shakespeare's play *Romeo and Juliet*. The URN for that edition would be *urn:isbn:0-486-27557-4*. However, it gives no information as to where to find a copy of that book.
- URI = scheme:[//authority]path[?query][#fragment]
- Examples of scheme: https, ldap, mailto, tel, ftp
- Un IRI è una forma generale di Uniform Resource Identifier costituita, a differenza di una URI, da una sequenza di caratteri appartenenti all'*Universal Character Set*

Various types of literals

- (ex:thisLecture ex:title "graph databases") - untyped
- (ex:thisLecture ex:title "graph databases"@en) - untyped, assigned with "English" (en) language
- (ex:thisLecture ex:title   "graph databases"^^xsd:string) - explicit type string
- Other types:
  - xsd:integer
  - xsd:decimal
  - xsd:float
  - xsd:boolean
  - xsd:date
  - xsd:time

Vocabularies

The **RDF built-in vocabulary** assigns a specific meaning to certain terms, which are the URIs having the prefix
`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
(usually abbreviated as `rdf:`)

Some examples:

`rdf:type` `rdf:subject` `rdf:predicate`

`rdf:object` `rdf:Statement` `rdf:Property ...`

Other popular vocabularies exist, such as

– Dublin Core (`dc:` or `dcterms:`) – `http://purl.org/dc/terms/`

– FOAF (`foaf:`) - `http://xmlns.com/foaf/0.1/`

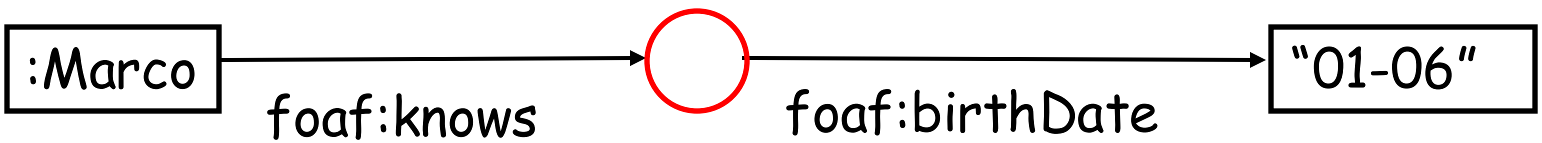
– RDFS (`rdfs:`) – `https://www.w3.org/2000/01/rdf-schema`

You may see them as simple namespaces, or systems of metadata

Blank nodes: unidentifiable resources

blank node (bnode) = RDF graph node with "anonymous label" (i.e., not associated with an URI)

Example: Marco knows someone which was born on the Epiphany day




```
Marco   foaf:knows       _:X.
_:X     foaf:birthDate   "01-06".
```

Exercise

- Write an RDF dataset expressing the following facts
  - John is friend of Mary
  - Mary is friend of Ann since ‘2019-03-12’

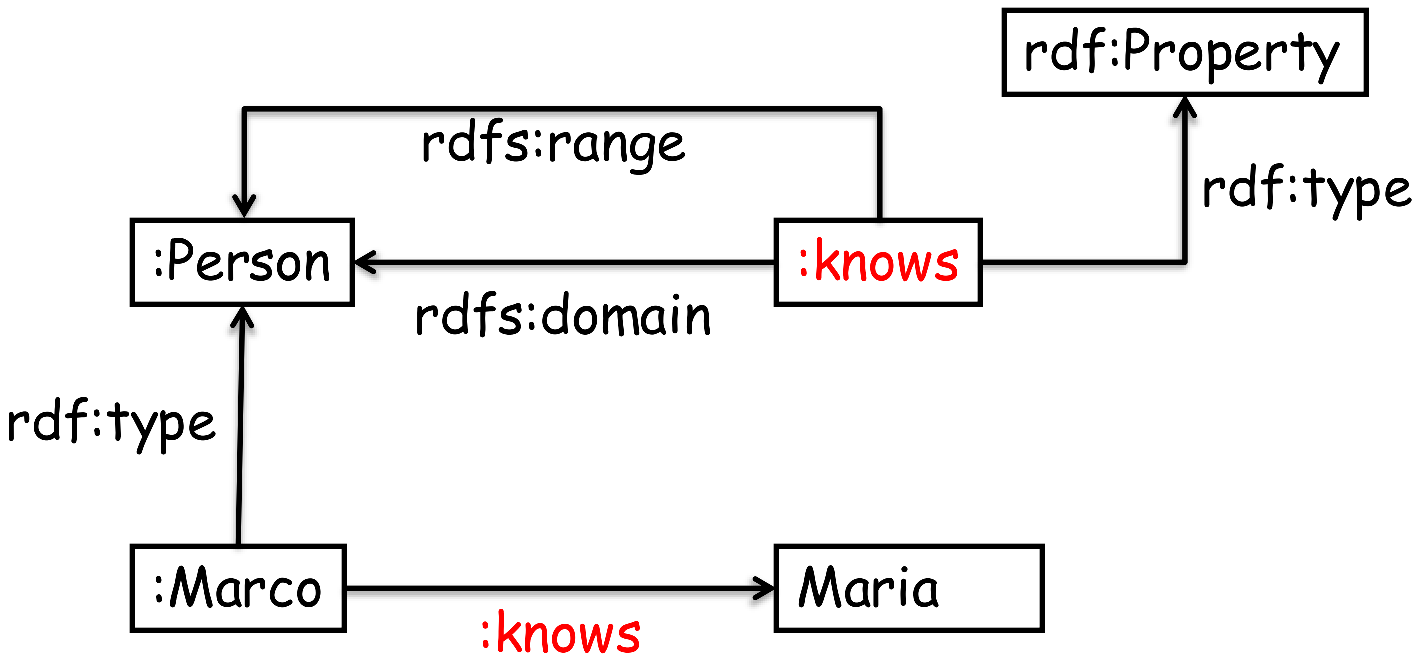
  Use the URI that you like (you can assume an empty prefix).

Solution

```
:John :name 'John' .
:Mary :name 'Mary' .
:Ann :name 'Ann' .
:John :isFriendOf :Mary .
:Mary :hasFriendship _:x .
:Ann :isInFriendship _:x .
_:x :since '2019-03-12' .
```

URIs both as predicate and subject

Example: Marco is a person, Marco knows Maria, knows is a property between pairs of people



Higher-order statements

- One can make RDF statements about other RDF statements
  - example: “Ralph believes that the web contains one billion documents”
- Higher-order statements
  - allow us to express beliefs (and other modalities)
  - are important for trust models, digital signatures, etc.
  - also: metadata about metadata
  - are represented by modeling RDF in RDF itself

$\Rightarrow$ basic tool: reification, i.e., representation of an RDF triple as a resource

Reification

Reification in RDF = using an RDF statement as the subject (or object) of another RDF statement

Examples of statements that need reification to be expressed in RDF:

- "The New York Times claims that Joe is the author of the book ABC"
- "The statement "The Divina Commedia was written by Dante Alighieri" was written by the British National Library"

Reification

- RDF provides a built-in predicate vocabulary for reification:
  - **rdf:subject**
  - **rdf:predicate**
  - **rdf:object**
  - **rdf:statement**
- Using this vocabulary (i.e., these URIs from the rdf: namespace) it is possible to represents a triple through a blank node

Reification: example

- the statement "Joe is the author of the book ABC" can be represented by the following triples:

```
_:x rdf:type rdf:statement.
_:x rdf:predicate dc:creator.
_:x rdf:subject :ABC.
_:x rdf:object "Joe".
```

- The blank node _:x is the **reification** of the statement (it is an anonymous URI that represents the whole triple). Notice that the above set of triples has the same intuitive meaning of

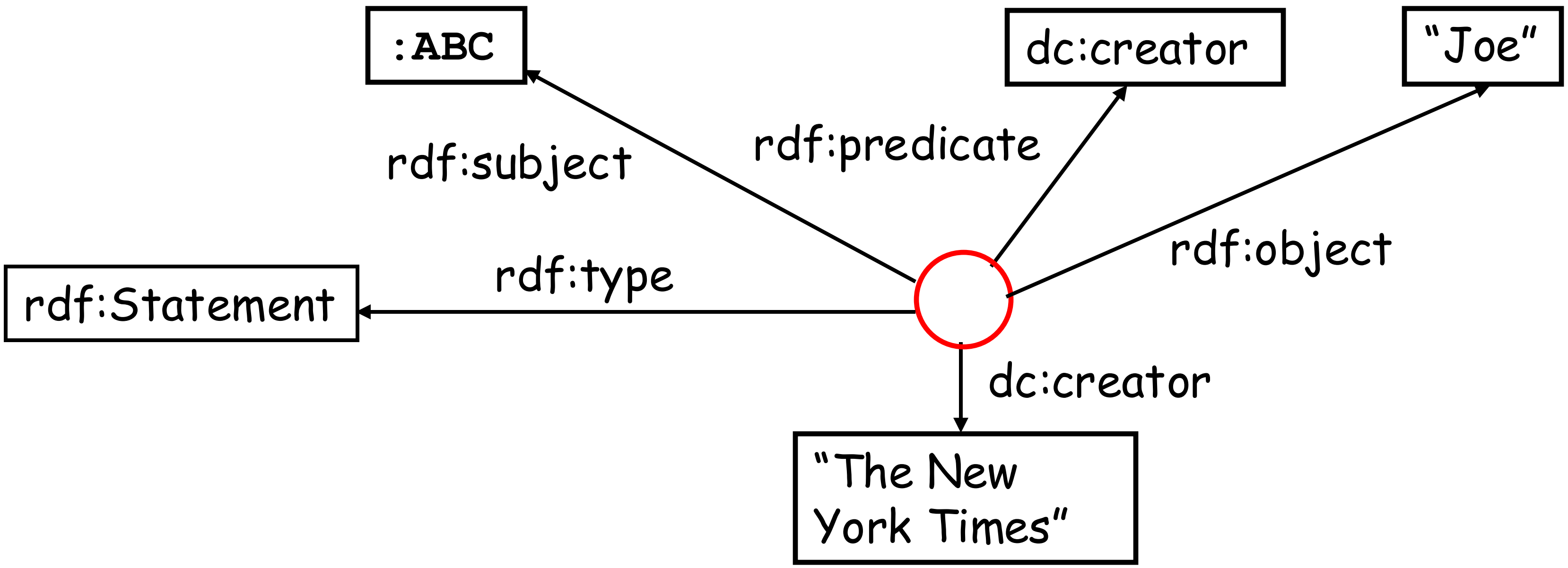
```
:ABC dc:creator "Joe"
```

- Now, "the New York Times claims that Joe is the author of book ABC" can be represented using the bnode _:x, by adding to the above four triples the following triple:

```
_:x dc:creator "The New York Times".
```

Reification: example

The New York Times claims that *Joe is the author of the book ABC*



Exercise

- Write an RDF dataset expressing the following fact
  - John believes that Mary is friend of Ann since ‘2019-03-12’

  Use the URI that you like (you can assume an empty prefix).

Solution

```
_:x rdf:type rdf:statement.
_:x rdf:predicate :isFriendOf.
_:x rdf:subject :Mary.
_:x rdf:object :Ann.
:John :believes _:x
_:x :since '2019-03-12'
```

Is this a good solution?

Solution (2)

```
_:x rdf:type rdf:statement.
_:x rdf:predicate :hasFriendship.
_:x rdf:subject :Mary.
_:x rdf:object _:z.
_:y rdf:type rdf:statement.
_:y rdf:predicate :isInFriendship.
_:y rdf:subject :Ann.
_:y rdf:object _:z.
_:w rdf:type rdf:statement
_:w :predicate :since.
_:w :subject _:z.
_:w :object '2019-03-12'.
:John :believes _:w.
```

RDF syntaxes

RDF model = edge-labeled graph = set of triples

- graphical notation (graph)
- (informal) triple-based notation

  e.g., **`(subject, predicate, object)`**
- formal syntaxes:
  - N3 notation
  - Turtle notation
  - N-Triples
  - concrete (serialized) syntax: RDF/XML syntax

RDF syntaxes

- N3 (Notation3) is designed to be human-readable (if compared with the XML RDF syntax). N3 has some features that go beyond RDF (e.g., allows for the specification of RDF-based rules). Turtle is the subset of N3 specific for RDF (thus we focus on Turtle only)

- Turtle (Terse RDF Triple Language) notation.

Example:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix : <http://myPrefix/>.
:mark rdf:type :bankclerk.
:mark :worksFor :NationalBank.
:mark :name "Marco"@it.
_:x :worksFor :NationalBank.
_:x :idNumber "54321"^^xsd:integer.
```

symbols preceded by _: denote blanks

Turtle Notation: Example*

```
@prefix dc: <http://purl.org/dc/terms> .
@prefix ex: <http://example.org/> .
<http://www.w3.org/TR/rdf-syntax-grammar>
  dc:title "RDF 1.1 XML Syntax" ;
  ex:editor [
    ex:fullname "Fabien Gandon";
    ex:homePage <http://www-sop.inria.fr/members/Fabien.Gandon>
       ] .
```

The example encodes an RDF database that expresses the following facts:

- The W3C technical report on RDF syntax and grammar has the title “RDF 1.1 XML Syntax”.

- That report's editor is a certain individual, who in turn

  - has full name Fabien Gandon.

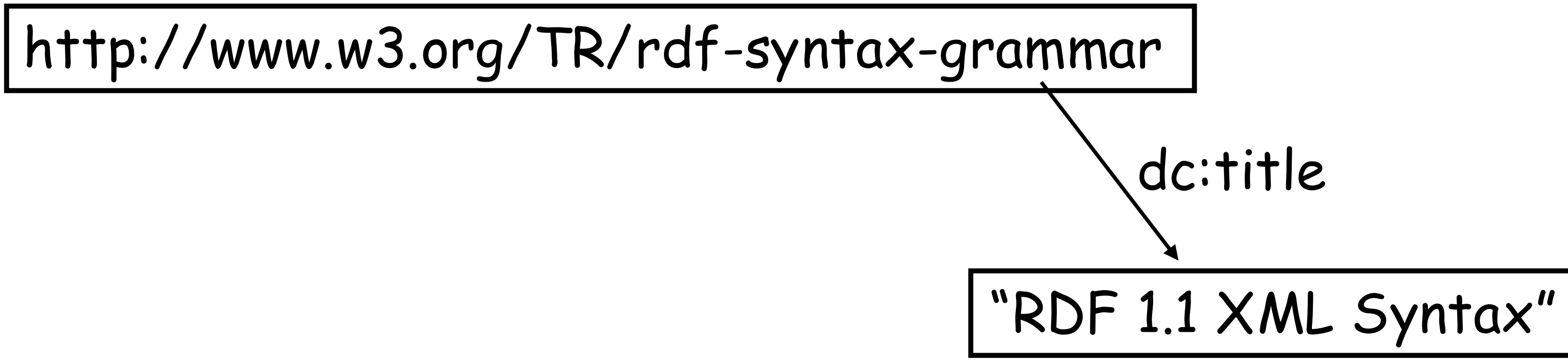
  - has a home page at http://www-sop.inria.fr/members/Fabien.Gandon.

*Adapted from http://en.wikipedia.org/wiki/Turtle_(syntax)

RDF/XML syntax

- A node that represents a resource (labeled or not) is represented by an XML element whose XML type is `rdf:Description`, while its label, if any, is defined as the value of the `rdf:about` property
- An edge outgoing from a node N is represented as a sub-element of the element that represents N. The XML type of this sub-element is the label of the edge.
- The end node of an edge is represented as the content of the element representing the edge. It is either a
  - a value (if the end node contains a literal)
  - or a new resource (if the end node contains a URI): in this case it is represented by a sub-element whose XML type is `rdf:Description`
- Values (literals) can be assigned with an XML type (the same defined in XML-Schema)
- Prefixes can be declared by using the standard XML attribute to refer to name spaces, i.e., **`xmlns`** (tipically this is done in the document root element)

RDF/XML syntax: Example




```
<?xml version=“1.0”?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/terms"  >
  <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
        <dc:title>RDF 1.1 XML Syntax</dc:title>
   </rdf:Description>
</rdf:RDF>
```

RDF Storage*

- RDF data management has been studied in a variety of contexts. This variety is actually reflected in a richness of the perspectives and approaches to storage and indexing of RDF datasets, typically driven by particular classes of query patterns and inspired by techniques developed in various research communities.
- In the literature, we can identify three main basic perspectives underlying this variety.
  - The relational perspective.
  - The entity perspective.
  - The pure graph-based perspective.

*From: *Storing and Indexing Massive RDF Data Sets.* Yongming Luo, Francois Picalausa, George H.L. Fletcher, Jan Hidders, and Stijn Vansummeren. In Semantic Search over the Web. Springer. 2012

The relational perspective

- An RDF graph is seen just as a particular type of relational data, and techniques developed for storing, indexing and answering queries on relational data can hence be reused and specialized for storing and indexing RDF graphs.
- The most naive approach in this respect is simply to store all RDF triples in a single table over the relation schema *(subject, predicate, object)*. Some implementations include an additional context column in order to store more than one single RDF graph. In this case, the context column specifies the IRI of the named graph in which the RDF triple occurs.
- This kind of representation is known as the vertical representation

The relational perspective – Vertical representation

- Due to the large size of the RDF graphs and the potentially large number of self-joins required to answer queries, care must be taken to devise an efficient physical layout with suitable indexes to support query answering.

- Unclustered BTree indexes: define BTree indexes on the triple table (s,p,o). Four different sets of indexes are usually adopted:
  - an index on the subject column (s) alone; an index on the property (p) column alone, and index on the object column (o) alone.
  - a combined index on subject and property (sp), as well as an index on the object column (o) alone.
  - a combined index on property and object (po).
  - a combined clustered index on all columns together (spo).

- Clustered BTree indexes: store various sorted versions of the triple store table according to various permutation of the sequence s,p,o, over which define indexes allowing for fast access (even though they require more space and management of redundant information).

The relational perspective – Horizontal representation

- A different approach under the relational perspective provides a horizontal representation of RDF

- According to such representation, data are conceptually stored in a single table that has one column for the subject and one column for each predicate that occurs in the RDF graph. Then, it has one row for each subject, and for each (s,p,o) triple, the object o is placed in the p column of row s.

The horizontal representation - example

rdf triples

$$
\begin{array}{lll}
\{\langle \text{work5678}, & \text{FileType}, & \text{MP3} \rangle, \\
\langle \text{work5678}, & \text{Composer}, & \text{Schoenberg} \rangle, \\
\langle \text{work1234}, & \text{MediaType}, & \text{LP} \rangle, \\
\langle \text{work1234}, & \text{Composer}, & \text{Debussy} \rangle, \\
\langle \text{work1234}, & \text{Title}, & \text{La Mer} \rangle, \\
\langle \text{user8604}, & \text{likes}, & \text{work5678} \rangle, \\
\langle \text{user8604}, & \text{likes}, & \text{work1234} \rangle, \\
\langle \text{user3789}, & \text{name}, & \text{Umi} \rangle, \\
\langle \text{user3789}, & \text{birthdate}, & \text{1980} \rangle, \\
\langle \text{user3789}, & \text{likes}, & \text{work1234} \rangle, \\
\langle \text{user8604}, & \text{name}, & \text{Teppei} \rangle, \\
\langle \text{user8604}, & \text{birthdate}, & \text{1975} \rangle, \\
\langle \text{user8604}, & \text{phone}, & \text{2223334444} \rangle, \\
\langle \text{user8604}, & \text{phone}, & \text{5556667777} \rangle, \\
\langle \text{user8604}, & \text{friendOf}, & \text{user3789} \rangle, \\
\langle \text{Debussy}, & \text{style}, & \text{impressionist} \rangle, \\
\langle \text{Schoenberg}, & \text{style}, & \text{expressionist} \rangle, \ldots \}
\end{array}
$$

relational horizontal representation

subject	FileType	Composer	...	phone	friendOf	style
work5678	MP3	Schoenberg				
work1234		Debussy				
...						
user8604				{2223334444, 5556667777}	user3789	
Debussy						impressionist
Schoenberg						expressionist

The horizontal representation

- As can be seen from the previous example, it is uncommon that a subject occurs with all possible predicate values, leading to sparse tables with many empty cells. Care must hence be taken in the physical layout of the table in order to avoid storing the empty cells.

- Also, since it is possible that a subject has multiple objects for the same predicate (e.g., user8604 has multiple phone numbers), each cell of the table represents in principle a set of objects, which again must be taken into account in the physical layout.

The horizontal representation – property tables

- To minimize the storage overhead caused by empty cells, the so-called property-table approach concentrates on dividing the wide table in multiple smaller tables containing related predicates
- For example, in the music fan RDF graph, different tables could be introduced for Works, Fans, and Artists. In this scenario, the Works table would have columns for Composer, FileType, MediaType, and Title, but would not contain the unrelated phone or friendOf columns.
- How to divide the wide table into property tables is up to the designers (support for this is provided by some RDF tools)

The horizontal representation – vertical partitioning

- The so-called vertically partitioned database approach (not to be confused with the vertical representation approach) takes the decomposition of the horizontal representation to its extreme:

  each predicate column *p* of the horizontal table is materialized as a binary table over the schema *(subject, p)*. Each row of each binary table corresponds to a triple.

- Note that, hence, both the empty cell issue and the multiple objects issue are solved at the same time.

The relational perspective – storage of URIs and literals

- Indipendently from the approach followed, under the relational storage of RDF graphs a certain policy is commonly addressed on how to store values in tables: rather than storing each URI or literal value directly as a string, implementations usually associate a unique numerical identifier to each resource and store this identifier instead. Indeed,
  - since there is no a priori bound on the length of the URIs or literal values that can occur in RDF graphs, it is necessary to support variable-length records when storing resources directly as strings
  - RDF graphs typically contain very long URI strings and literal values that, in addition, are frequently repeated in the same RDF graph.

- Unique identifiers can be computed in two general ways:

  (i) applying a hash function to the resource string;

  (ii) maintaining a counter that is incremented whenever a new resource is added. In both cases, dictionary tables are used to translate encoded values into URIs and literals

The entity perspective for storing RDF graphs

The second basic perspective, originating from the information retrieval community, is the entity perspective:

- Resources in the RDF graph are interpreted as "objects", or "entities"
- each entity is determined by a set of attribute-value pairs
- In particular, a resource *r* in RDF graph **G** is viewed as an entity with the following set of (attribute,value) pairs:

$$entity(r) = \{(p, o) \mid (r, p, o) \in G\} \cup \{(p^{-1}, o) \mid (o, p, r) \in G\}.$$

The entity perspective - example

{⟨work5678,	FileType,	MP3 ⟩,
⟨work5678,	Composer,	Schoenberg ⟩,
⟨work1234,	MediaType,	LP ⟩,
⟨work1234,	Composer,	Debussy ⟩,
⟨work1234,	Title,	La Mer ⟩,
⟨user8604,	likes,	work5678 ⟩,
⟨user8604,	likes,	work1234 ⟩,
⟨user3789,	name,	Umi ⟩,
⟨user3789,	birthdate,	1980 ⟩,
⟨user3789,	likes,	work1234⟩,
⟨user8604,	name,	Teppei ⟩,
⟨user8604,	birthdate,	1975 ⟩,
⟨user8604,	phone,	2223334444 ⟩,
⟨user8604,	phone,	5556667777 ⟩,
⟨user8604,	friendOf,	user3789 ⟩,
⟨Debussy,	style,	impressionist⟩,
⟨Schoenberg,	style,	expressionist⟩, . . . }

rdf triples

entity view

```
work5678
FileType: MP3
Composer: Schoenberg
likes^-1 : user8604
```

```
work1234
Title     : La Mer
MediaType: LP
Composer : Debussy
likes^-1  : user8604
```

```
user3789
name       : Umi
birthdate : 1980
likes      : work1234
friendOf^-1: user8604
```

```
user8604
name      : Teppei
birthdate: 1975
phone     : 2223334444
phone     : 5556667777
likes     : work5678
likes     : work1234
friendOf  : user3789
```

```
Debussy
style       : impressionist
Composer^-1: work5678
```

```
Schoenberg
style       : expressionist
Composer^-1: work1234
```

The entity perspective

- Techniques for information retrieval can then be specialized to support query patterns that retrieve entities based on particular attributes and/or values.

- For example, in the previous representation we have that the entity user8604 is retrieved when searching for entities born in 1975 (i.e., have 1975 as a value on the attribute birthdate) as well as when searching for entities with friends who like Impressionist music. These are examples of entity-centric queries, whose aim is to return entire entities satisfying some conditions. Also keyword queries are normally supported by tools adopting the entity perspective.

- Specific tools provide peculiar solutions to these problems.

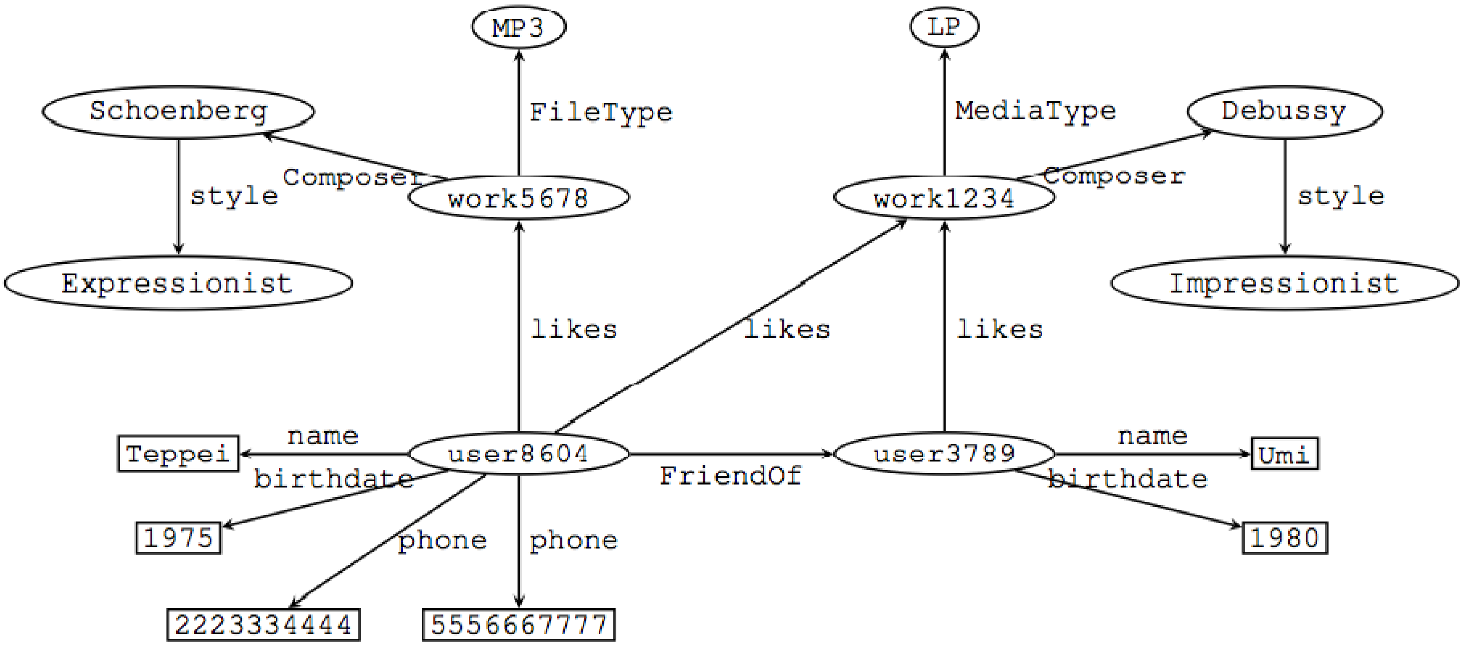
The graph-based perspective for storing RDF graphs

- Under this graph-based perspective, the focus is on supporting navigation in the RDF graph when viewed as a classical graph in which subjects and objects form the nodes, and triples specify directed, labeled edges. The aim is therefore to natively store RDF dataset as graphs.

- Typical queries supported in this perspective are graph-theoretic queries such as reachability between nodes, or path expressions, e.g., check if there is a certain type of path between two nodes.

- The major issue under this perspective is how to explicitly and efficiently store and index the implicit graph structure.

The graph-based perspective for storing RDF graphs



Querying RDF: SPARQL

Simple Protocol And RDF Query Language

- W3C standardisation effort similar to the XQuery query language for XML data
- Data Access Working Group (DAWG)
- Suitable for remote use (remote access protocol)

SPARQL – query structure

SPARQL query includes, in the following order:

- prefix declaration, to abbreviate URIs (optional)
- SELECT clause, to specify the information to be returned
- dataset definitions, to specify the URI of the graph to be queried, possibly more than one
- WHERE clause, to specify the query pattern, i.e., the conditions that have to be satisfied by the triples of the dataset
- additional modifiers, to re-organize the results of the query (optional)

```
# prefix declaration
PREFIX es: <...>
...
# data to be returned
SELECT ...
# dataset definition
FROM <...>
# graph pattern specification
WHERE { ...}
# modifiers
ORDER BY ...
```

SPARQL – the WHERE clause

- The `WHERE` clause contains a basic graph pattern (BGP), consisting of:
  - a set of triples separated by "."
    - "." has the semantics of the AND
    - object, predicate and/or subject can be variables
- It also possibly contains:
  - a `FILTER` a condition that, using Boolean expressions, specifies some constraints that must be satisfied by the tuples in the result;
  - an `OPTIONAL` condition that indicates a pattern that may (but does not need to) be satisfied by a subgraph, to produce a tuple in the result;
  - other operators (e.g., `UNION`)

SPARQL – a first example

```
SELECT ?s ?p ?o
FROM <mygraph.db>
WHERE {
   ?s ?p ?o
}
```

Returns all the triples in the dataset

- Variables are indicated through the "`?`" prefix ("`$`" is also possible)
- Subject, predicate and object are all variables in this example
- In this example, the SELECT clause can be also equivalently written as: `SELECT *`

SPARQL – more complex example

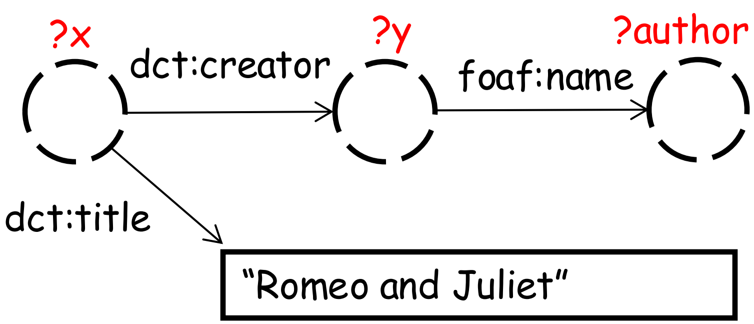
```
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?author
FROM <http://bnb.data.bl.uk/id/data/BNB>
WHERE { ?x dct:creator ?y.
        ?x dct:title "Romeo and Juliet".
        ?y foaf:name ?author}
```

- The `?author` variable denotes the result to be returned.
- The `FROM` clause in this case specifies the URI of the RDF graph of the British National Library
- The SPARQL query processor returns all hits matching the pattern of the four RDF-triples.

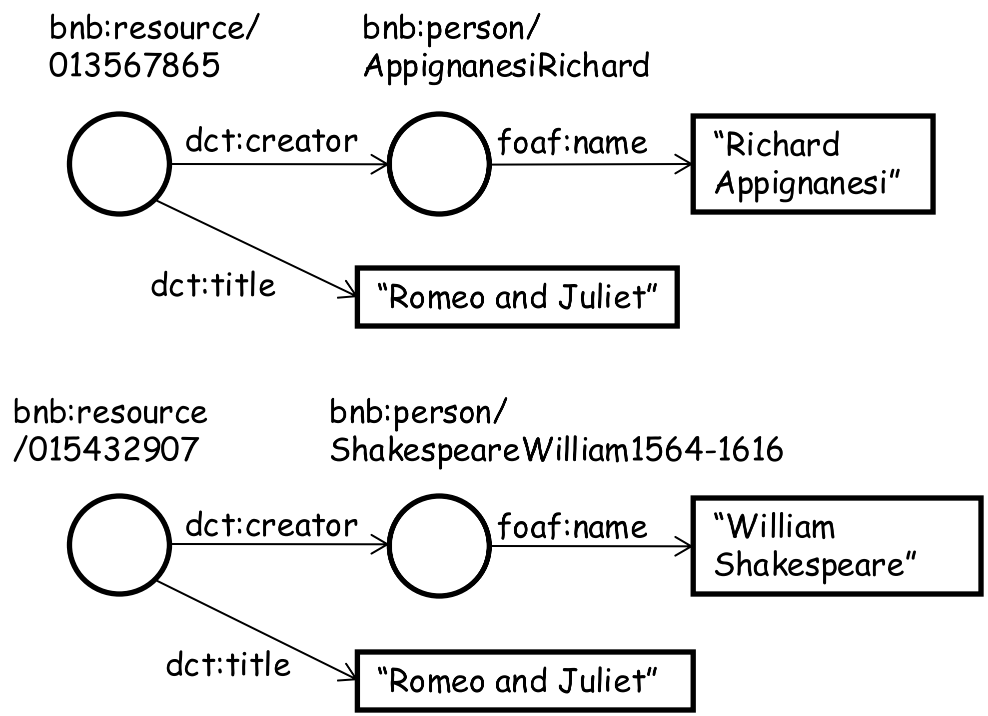
SPARQL – query evaluation

The query returns all values A for which there exist resources R1, R2, such that the triples obtained by replacing variables `?authors`, `?x` and `?y`, with A, R1 and R2, respectively, occur in the queried graph.

QUERY



QUERIED GRAPH



RESULT

author
"Richard Appignanesi"
"William Shakespeare"

Note: `bnb = bnb.data.bl.uk/doc/`

SPARQL – semantics of Basic Graph Pattern

We have already seen that a Basic Graph Pattern in SPARQL can be seen as a graph (set of triples), but with variables.

Let G be graph and let B be a Basic Graph Pattern in SPARQL. A **homomorphism** h from B to G is a function from the components of triples of B to the resources of G such that:

- h(c) = c, for every URI c (constants are mapped to themselves),
- h(x) = d, for every variable x in B (where d in a resources in G),
- for every $(\alpha\ \beta\ \gamma)$ in B, we have that $(h(\alpha)\ h(\beta)\ h(\gamma))$ is in G.

Let Q be a SPARQL query with target list $?x_1\ ?x_2\ ...\ ?x_m$ and Basic Graph Pattern B. The answer of Q with respect to G is the (multi)set:

$\{ <h(?x_1)\ h(?x_2)\ ...\ h(?x_n)> \mid h \text{ is a homomorphism from B to G} \}$

SPARQL endpoints

- SPARQL queries are executed on RDF dataset
- A SPARQL endpoint accepts queries and returns results via the HTTP protocol
  - generic endpoints query all RDF datasets that are accessible via the Web
    - `http://lod.openlinksw.com/sparql`
  - dedicated endpoints are intended to query one or more specific dataset
    - `http://bnb.data.bl.uk/doc/data/BNB` (or `https://bnb.data.bl.uk/sparql`)
    - `http://dbpedia.org/sparql`
    - `https://query.wikidata.org/`
- The `FROM` clause, in principle, is mandatory, but
  - when the endpoint is dedicated you can omit it in the specification of queries over such endpoint
  - when the endpoint is generic, there is often a default dataset that is queried in the case in which the `FROM` clause is not specified
- In our examples, we often omit the `FROM` clause, implicitly assuming we are querying specific endpoints

SPARQL results

- The result of a query is a set of tuples, whose structure (labels and cardinality) reflects what has been specified in the `SELECT` clause
- The SPARQL endpoint typically allows one to indicate the syntax for the result
  - HTML
  - XML
  - CSV
  - JSON
  - XML/RDF
  - Turtle
  - N-Triples
  - ….
- Notice that the result is always a set of triples, but encoded in different syntaxes, as selected by the user.

Query over a generic endpoint*

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX card: <http://www.w3.org/People/Berners-Lee/card#>
SELECT ?homepage
FROM <http://www.w3.org/People/Berners-Lee/card>
WHERE {
    card:i foaf:knows ?known .
    ?known foaf:homepage ?homepage .
}
```

Execute the above query on the generic end point

http://lod.openlinksw.com/sparql

Note: card:i is the URI referring to Tim Berners-Lee within his RDF FOAF file

*From: https://www.w3.org/2009/Talks/0615-qbe/

Query over a specific endpoint

```
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?author
WHERE { ?x dct:creator ?y.
        ?x dct:title "Romeo and Juliet".
           ?y foaf:name ?author}
```

Execute the above query on the dedicated endpoint of the British National Library (note the absence of the FROM clause). As a suggestion, use the flint editor version

https://bnb.data.bl.uk/flint-sparql

SPARQL query – BGP example

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlow@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
_:c foaf:mbox <mailto:carol@example.org> .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE { ?x foaf:name ?name .
        ?x foaf:mbox ?mbox }
```

result:

“Johnny Lee Outlaw”	<mailto:jlow@example.com>
“Peter Goodguy”	<mailto:peter@example.com>

Execute on GraphDB (local installation)

SPARQL – use of FILTER: example

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlow@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
_:c foaf:mbox <mailto:carol@example.org> .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE { ?x foaf:name ?name .
        ?x foaf:mbox ?mbox .
        FILTER regex(?name, "^J") }
```

begins with J

result:

“Johnny Lee Outlaw”	<mailto:jlow@example.com>

Execute on GraphDB (local installation)

Predicates that can be used in the FILTER clause

- Logical connectives:
  ! (NOT)
  && (AND)
  || (OR)
- Comparison: >, <, =, != (not equal),
- Functions:
  - generic: IN, NOT IN, BOUND, LANG,….
  - on strings (REGEX, CONCAT, UCASE, LOCASE,..),
  - on numerics (ROUND, ABS,..),
  - on dates and times (YEAR, DAY, NOW,..)
- Test: isURI, isBlank, isLiteral, isNumeric, ...

SPARQL – use of FILTER: example

Add to the RDF graph:

```
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
_:a dbo:birthDate "1976-10-11"^^xsd:date .
_:b dbo:birthDate "1986-09-23"^^xsd:date .
_:c dbo:birthDate "1996-01-07"^^xsd:date .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?name ?date
WHERE { ?x foaf:name ?name .
        ?x dbo:birthDate ?date
        FILTER (?date >= "1970-01-01"^^xsd:date &&
                      ?date < "1980-01-01"^^xsd:date) }
```

result:

“Johnny Lee Outlaw”	"1976-10-11"^^xsd:date

Execute on GraphDB (local installation)

SPARQL – optional patterns: example

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlow@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
_:c foaf:mbox <mailto:carol@example.org> .
.....
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE { ?x foaf:mbox ?mbox .
        OPTIONAL {?x foaf:name ?name } }
```

result:

“Johnny Lee Outlaw”	<mailto:jlow@example.com>
“Peter Goodguy”	<mailto:peter@example.com>
	<mailto:carol@example.org>

Execute on GraphDB (local installation)

SPARQL – optional patterns: example 2

- Return all resources contained in the dataset of the British National Bibliography, whose title is "Romeo and Juliet", along with the 10-digits ISBN and the 13-digits ISBN, if they have them

```
prefix dct:<http://purl.org/dc/terms/>
prefix bibo:<http://purl.org/ontology/bibo/>
select ?x ?i10 ?i13
WHERE {?x dct:title "Romeo and Juliet".
       OPTIONAL {?x bibo:isbn10 ?i10.
                 ?x bibo:isbn13 ?i13}}
```

- Run the query on the SPARQL end point of the British national digital library (http://bnb.data.bl.uk/doc/data/BNB) and compare the results obtained with those returned by the version of the query where `?i10` and `?i13` are not optional.

SPARQL – Negation (as failure): example

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:knows _:b .
_:b foaf:knows _:c .
_:a foaf:knows _:c .
_:f foaf:knows _:a .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x
WHERE {
       ?x foaf:knows ?y .
        OPTIONAL{?y foaf:knows ?z} .
         FILTER(!BOUND(?z))
 }
```

result:

_:a
_:b

This query returns resources that know someone that does not know anyone

Execute on GraphDB (local installation)

SPARQL – Negation (as failure): alternative

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:knows _:b .
_:b foaf:knows _:c .
_:a foaf:knows _:c .
_:f foaf:knows _:a .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x
WHERE {
       ?x foaf:knows ?y .
        FILTER NOT EXISTS{?y foaf:knows ?z} .
 }
```

result:

_:a
_:b

This query returns resources that know someone that does not know anyone

Execute on GraphDB (local installation)

SPARQL – Negation: example 2

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:knows _:b .
_:b foaf:knows _:c .
_:a foaf:knows _:c .
_:f foaf:knows _:a .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x
WHERE {
       ?x foaf:knows ?y .
        FILTER NOT EXISTS {?x foaf:knows ?z .
                           ?z foaf:knows ?w }
}
```

result:

_:b

This query returns resources that know someone and know **only** resources that do not know anyone

Execute on GraphDB (local installation)

Negation through MINUS: example 2

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:knows _:b .
_:b foaf:knows _:c .
_:a foaf:knows _:c .
_:f foaf:knows _:a .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x
WHERE {
       ?x foaf:knows ?y .
        MINUS {?x foaf:knows ?z .
               ?z foaf:knows ?w }
}
```

result:

_:b

This query returns resources that know someone and know **only** resources that do not know anyone

Execute on GraphDB (local installation)

Property paths: example

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:knows _:b .
_:b foaf:knows _:c .
_:a foaf:knows _:c .
_:f foaf:knows _:a .
_:c foaf:knows _:d .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x ?y
WHERE { ?x foaf:knows+ ?y }
```

result:

_:a	_:b
_:b	_:c
_:a	_:c
_:f	_:a
_:c	_:d
_:a	_:c
_:a	_:d
..	..

This query compute the transitive closure of the foaf:knows relation

Execute on GraphDB (local installation)

Path language

Syntax Form	Matches
uri	A URI or a prefixed name. A path of length one.
^*elt*	Inverse path (object to subject).
(*elt*)	A group path *elt*, brackets control precedence.
elt1 / *elt2*	A sequence path of *elt1*, followed by *elt2*
elt1 ^ *elt2*	Shorthand for *elt1* / ^*elt2*, that is *elt1* followed by the inverse of *elt2*.
elt1 | *elt2*	A alternative path of *elt1*, or *elt2* (all possibilities are tried).
*elt**	A path of zero or more occurrences of *elt*.
elt+	A path of one or more occurrences of *elt*.
elt?	A path of zero or one *elt*.
elt{*n*,*m*}	A path between n and m occurrences of *elt*.
elt{*n*}	Exactly *n* occurrences of *elt*. A fixed length path.
elt{*n*,}	*n* or more occurrences of *elt*.
elt{,*n*}	Between 0 and *n* occurrences of *elt*.

uri is either a URI or a prefixed name and *elt* is a path element, which may itself be composed of path syntax constructs.

From: https://www.w3.org/TR/sparql11-property-paths/

SPARQL: UNION - example

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:knows _:b .
_:b foaf:knows _:c .
_:a foaf:knows _:c .
_:c foaf:knows _:d .
_:f foaf:knows _:a
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x ?y
WHERE {{ ?x foaf:knows ?y }
      UNION
      { ?y foaf:knows ?x }}
```

result:

_:a	_:b
_:a	_:c
_:b	_:c
_:c	_:d
_:f	_:a
_:b	_:a
_:c	_:a
_:c	_:b
_:d	_:c
_:a	_:f

Execute on GraphDB (local installation)

SPARQL – UNIONs of graph patterns

Example: Return all the resources stored in the dataset of the British National Bibliography, whose title is "Romeo and Juliet" and have either a 10-digits ISBN or a 13 digits ISBN

```
prefix dct:<http://purl.org/dc/terms/>
prefix bibo:<http://purl.org/ontology/bibo/>
select ?x ?i ?y
WHERE {{?x dct:title "Romeo and Juliet".
       ?x bibo:isbn10 ?i} UNION
       {?x dct:title "Romeo and Juliet".
       ?x bibo:isbn13 ?y}}
```

Run the query on the SPARQL end point of the British national digital library (http://bnb.data.bl.uk/doc/data/BNB)

SPARQL: AGGREGATION - example

RDF graph:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:knows _:b .
_:b foaf:knows _:c .
_:a foaf:knows _:c .
_:f foaf:knows _:a .
_:c foaf:knows _:d .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x (count(?y) AS ?count)
WHERE { ?x foaf:knows ?y .}
GROUP BY ?x
```

result:

_:a	"2"^^xsd:integer
_:b	"1"^^xsd:integer
_:c	"1"^^xsd:integer
_:f	"1"^^xsd:integer

Other useful aggregate operators: SUM, MIN, MAX, AVG,..

Execute on GraphDB (local installation)

SPARQL – Aggregation

Return the number of provinces:

```
PREFIX aci: <http://lod.aci.it/ontology/>
SELECT (count(distinct ?x) as ?count)
WHERE { ?x a aci:Province. }
```

It can be executed over this RDF dataset, which is the English version of the an analogous file available at http://lod.aci.it/

Execute on GraphDB (local installation)

SPARQL – Aggregation

For each province, return the number of cities, but only for those provinces with more than 150 cities:

```
PREFIX aci: <http://lod.aci.it/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?provinceName (count(distinct ?x) as ?count)
WHERE {
    ?x  a   aci:City.
    ?x aci:belongs_to_province ?p.
    ?p  rdfs:label ?provinceName
}
GROUP BY ?provinceName
HAVING (?count >150)
```


Execute on GraphDB (local installation)

SPARQL – “Querying predicates”

- In the graph pattern of a SPARQL query it is possible to label a predicate with a variable

- **Example**: which are the properties of the resource `<http://bnb.data.bl.uk/id/resource/015432907>`?

```
PREFIX bnb: <http://bnb.data.bl.uk/id/resource/>
SELECT DISTINCT ?p
WHERE {bnb:015432907 ?p ?v}
```

Run the query on the SPARQL end point of the British national digital library (http://bnb.data.bl.uk/doc/data/BNB)

SPARQL – example of query on DBPedia

- Return the names of all musical artists that were active in the sixties

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT DISTINCT ?name
WHERE { ?a rdf:type dbo:MusicalArtist .
        ?a rdfs:label ?name .
        ?a dbo:activeYearsEndYear ?dateEnd .
        ?a dbo:activeYearsStartYear ?dateStart
        FILTER (LANG(?name)="en" &&
               ?dateEnd >= "1960-01-01"^^xsd:date &&
               ?dateStart <"1970-01-01"^^xsd:date)
  }
```

Execute the query on the DBPedia endpoint (http://dbpedia.org/sparql)

Insert data through SPARQL

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
INSERT DATA
{
_:a dbo:birthDate "1976-10-11"^^xsd:date .
_:b dbo:birthDate "1986-09-23"^^xsd:date .
_:c dbo:birthDate "1996-01-07"^^xsd:date .
}
```

The effect on the database is *as* the RDF snippet (seen before):

```
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
_:a dbo:birthDate "1976-10-11"^^xsd:date .
_:b dbo:birthDate "1986-09-23"^^xsd:date .
_:c dbo:birthDate "1979-01-07"^^xsd:date .
```

Insert on the base of a query result

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbo: <http://dbpedia.org/ontology/>

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
INSERT
  {
        ?person dbo:birthPlace "NewYorkCity"^^xsd:string .
  }
WHERE
  {
        ?person foaf:name "Peter Goodguy" .
  }
```

We can use `DELETE` analogously.

CONSTRUCT

In SPARQL, the SELECT Clause can be substituted by a CONSTRUCT

The CONSTRUCT is a query form that is used to create an RDF graph through a user defined tempale

The result is an RDF graph obtained as follows

- All possible solutions of the BGP in the WHERE clause are considered
- The variables in the graph template are substituted with the values returned by the evaluation of the BGP in the WHERE clause
- The obtained triples are combined in a single RDF graph

CONSTRUCT

RDF graph:

```
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlow@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
_:c foaf:mbox <mailto:carol@example.org> .
```

query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT {?x foaf:name ?name .
           ?x foaf:mbox ?mbox }
WHERE { ?x foaf:name ?name .
        ?x foaf:mbox ?mbox }
```

result:

```
_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlow@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
```

CONSTRUCT: example

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
_:a rdf:type foaf:Person .
_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@example.com> .
_:a foaf:mbox <mailto:alice@work.example> .
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT {<http://example.org/person#Alice> vcard:FN ?name}
WHERE { ?x foaf:name ?name
}
```

```
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
<http://example.org/person#Alice> vcard:FN "Alice" .
```

Comparison with Neo4j Cypher

SPARQL	CYPHER
SELECT	RETURN
WHERE	MATCH
FILTER	WHERE (+ MATCH)
INSERT	CREATE
DELETE	DELETE
PATH LANGUAGE	PATH LANGUAGE (*;+;|;{n,m})
NOT EXISTS	NOT EXISTS
OPTIONAL	OPTIONAL MATCH
AGGREGATION	- (implicit)
-	PROPERTIES ON NODES
-	PROPERTIES ON EDGES
QUERYNG PREDICATES	-
CONSTRUCT	-
- (depend on the triple store)	INDEXES
- (schema mixed with data – e.g., through RDFS – can be queried in the same way as data)	SCHEMALESS (limited forms of contraints)

Exercise SPARQL 1

Write SPARQL queries that match the following requests:

1. return all URIs that have an author and creation date
2. return all predicates that have both URI1 and URI2 as subject
3. return all predicates that have either URI1 or URI2 as subject
4. return the name of all authors of any documents with a creation date

Exercise SPARQL 2

Write the SPARQL query that matches the following request:

1. return the name of the authors and the creation date of any document having an author and, optionally, a creation date

RDF/SPARQL tools

- **Jena** = Java framework for handling RDF models and SPARQL queries (http://jena.sourceforge.net/)
- **Virtuoso** = database system able to deal with RDF data and SPARQL queries, based on the use of an object-relational DBMS
(http://virtuoso.openlinksw.com/)
- Blazegraph (https://www.blazegraph.com/)
- Allegrograph (http://www.franz.com/agraph/allegrograph/)
- GraphDB (https://www.ontotext.com/products/graphdb/)
- …and many more, see
http://esw.w3.org/topic/SparqlImplementations

References

- The part on RDF storage is taken from: Y. Luo, F. Picalausa, G. H.L. Fletcher, J. Hidders, and S. Vansummeren. Storing and Indexing Massive RDF Data Sets. In Semantic Search over the Web. Springer. 2012

Additional bibliography:

- RDF 1.1 Concepts and Abstract Syntax - https://www.w3.org/TR/rdf11-concepts/
- RDF XML 1.1 Syntax https://www.w3.org/TR/rdf-syntax-grammar/
- SPARQL 1.1 Query Language https://www.w3.org/TR/sparql11-query/

Knowledge Graphs

A knowledge graph (KG) is a(n extension of) graph database whose aim is to represent the knowledge about a domain, rather than plain data. This implies that in the graph one should represent the semantics of the domain, describing classes, their instances and their relationships and the management system should be able to deal with incomplete information and to perform reasoning over the domain.

In order to make reasoning effective, knowledge graphs may make use of ontologies, often expressed in logic: they allow logical inference for retrieving implicit knowledge rather than only allowing queries requesting explicit knowledge.

The idea was made famous in 2012 by Google building on DBpedia and Freebase among other sources. Entity and relationship types associated with this knowledge graph have been further organized using terms from the schema.org vocabulary. The Google Knowledge Graph became a successful complement to string-based search within Google, and its popularity online brought the term into more common use.

Knowledge Graph as a set of RDF triples

Syntactically speaking, an RDF KG is simply a set of RDF triples.

Actually, we will illustrate the notion of RDFS-based KGs, rather than RDF KG. This means that we will use a specific vocabulary for labeling nodes and edges: the vocabulary of RDFS

RDF → RDFS

- RDFS originates as the schema language for RDF.
- The exact meaning of an RDF(S) graph was initially informally defined!
- Afterwards, a formal semantics has been provided using a translation to logic

  ⇒ the original formal definition of entailment and query answering over RDF(S) graphs has several problems!
- We will solve this problem by defining our own syntax and our formal semantics

RDFS-based Knowledge Graph

We will define an RDFS-based Knowledge Graph (KG in the following) in terms of:

- alphabet,

- syntax,

- semantics.

RDFS-based Knowledge Graph: alphabet

The alphabet of an RDFS-based KG includes:

- a fixed vocabulary, i.e., the following fixed set of "pre-defined" resource symbols (which are special URIs) denoting built-in predicates:
  - **`rdfs:Resource`**
  - **`rdfs:Class`**
  - **`rdf:type`**
  - **`rdfs:subClassOf`**
  - **`rdf:Property`**
  - **`rdfs:subPropertyOf`**
  - **`rdfs:domain`**
  - **`rdfs:range`**
- URIs denoting "user-defined" resources
- symbols denoting blank nodes

Note: **`type`** and **`Property`** already part of RDF vocabulary (cf. namespace rdf)

RDFS-based Knowledge Graph: alphabet

Every resource in a KG is an instance of the class **`rdfs:Resource`** and every instance of **`rdfs:Resource`** can play the role of:

- a pure individual (i.e. a resource that does not play the role of a class or a property, i.e., is not an instance of **`rdfs:Class`** or **`rdf:Property`**),
- a class (i.e, an instance of **`rdfs:Class`**), and
- a property (i.e., an instance of **`rdf:Property`**)

Moreover, by definition, **`rdfs:Resource`**, **`rdfs:Class`**, **`rdfs:Property`** play the role of a class in every KG, **`rdf:Type`**, **`rdfs:subClassOf`**, **`rdfs:subPropertyOf`**, **`rdfs:range`**, **`rdfs:domain`** play the role of a property in every KG.

RDFS-based Knowledge Graph: alphabet

- The built-in vocabulary of RDFS provides means for defining various aspects of classes and properties

- Vocabulary for classes, with the intended meaning:
  - **`rdfs:Resource`** the class of everything
  - **`rdfs:Class`**, the class of classes, that is used to assert that a resource is a class; note that **`rdfs:Class`** is a subclass of **`rdfs:Resource`**
  - **`rdf:type`**, used to assert that a resource is an instance of a class
  - **`rdfs:subClassOf`**, used to assert that a class is a subclass of another class

RDFS-based Knowledge Graph: alphabet

Vocabulary for properties, with the intended meaning:

- **`rdf:Property`**, the class of properties, used to assert that a resource is a property; **`rdfs:Property`** is a subclass of **`rdfs:Resource`**
- **`rdfs:domain`**, used to assert the type (class) of the first component of a property
- **`rdfs:range`**, used to assert the type (class) of the second component of a property
- **`rdfs:subPropertyOf`** used to assert that a property is a subproperty of another property

Note: **`rdf:Property`** already part of the RDF vocabulary (cf. namespace rdf)

RDFS-based Knowledge Graph: syntax

In our version of RDFS-based Knowledge Graph, we sanction that the possible triples forming a graph are of the following forms (where P is a user defined property):

- X **`rdf:type`** Y (class instance triple)
- X P Y (property instance triple)
- X **`rdfs:subClassOf`** Y (subclass triple)
- X **`rdfs:subPropertyOf`** Y (subproperty triple)
- X **`rdfs:domain`** Y (domain typing triple)
- X **`rdfs:range`** Y (range typing triple)

Triples of type (X **`rdf:type`** Y) and (X P Y) are called instance-based triples

RDFS-based Knowledge Graph: syntax

In our version of RDFS-based Knowledge Graph, we consider the following syntactic rules:

- a user-defined class is any user-defined resource that appears in at least one triple in a "class position", i.e., one of the following positions marked as ***X***:
  - ***X*** **`rdf:type rdfs:Class`**
    (***X*** *is an instance of* **`rdfs:Class`**, *and therefore is a class)*
  - $\boldsymbol{X_1}$ **`rdfs:subClassOf`** $\boldsymbol{X_2}$
    (*class* $\boldsymbol{X_1}$ *is a subclass of class* $\boldsymbol{X_2}$)
  - Y **`rdfs:domain`** ***X***
    (*the domain of the property Y is the class* ***X***)
  - Y **`rdfs:range`** ***X***
    (*the range of the property Y is the class* ***X***)
  - Y **`rdf:type`** ***X***
    (*individual Y is an instance of the class* ***X***)

RDFS-based Knowledge Graph: syntax

In our version of RDFS-based Knowledge Graph, we consider the following syntactic rules:

- a user-defined property is any user-defined resource that appears in at least one triple in a "property position", i.e., in one of the following positions marked as ***Z***:
  - ***Z*** **`rdf:type rdf:Property`**
    (***Z*** *is an instance of Property, and therefore is a property)*
  - Z_1 **`rdfs:subPropertyOf`** Z_2
    (*property* Z_1 *is a subproperty of property* Z_2)
  - ***Z*** **`rdfs:domain`** Y
    (*the domain of the property* ***Z*** *is the class Y)*
  - ***Z*** **`rdfs:range`** Y
    (*the range of the property* ***Z*** *is the class Y)*
  - Y_1 ***Z*** Y_2
    (*individual* Y_1 *is connected to individual* Y_2 *by the property* ***Z***)

RDFS-based KG: freedom in modeling

A class (i.e., an element that plays the role of a class in K) can also appear in a non-class position in K and a property (i.e., an element that plays the role of a property in K) can also appear in non-property position in K.

class position

class position

non class position

property position

a property not in a property position

non class position

```
Student rdfs:subClassOf Human.

Human rdf:type Species.

hasSupervisor rdf:type rdf:Property.

hasSupervisor concerns humanRelations

myrel rdfs:domain concerns

Student rdfs:domain myclass
```

RDFS-based KG: example

```
Frank rdf:type Student.
Jeen rdf:type Researcher.
Frank hasSupervisor Jeen.
```

RDFS-based KG: example

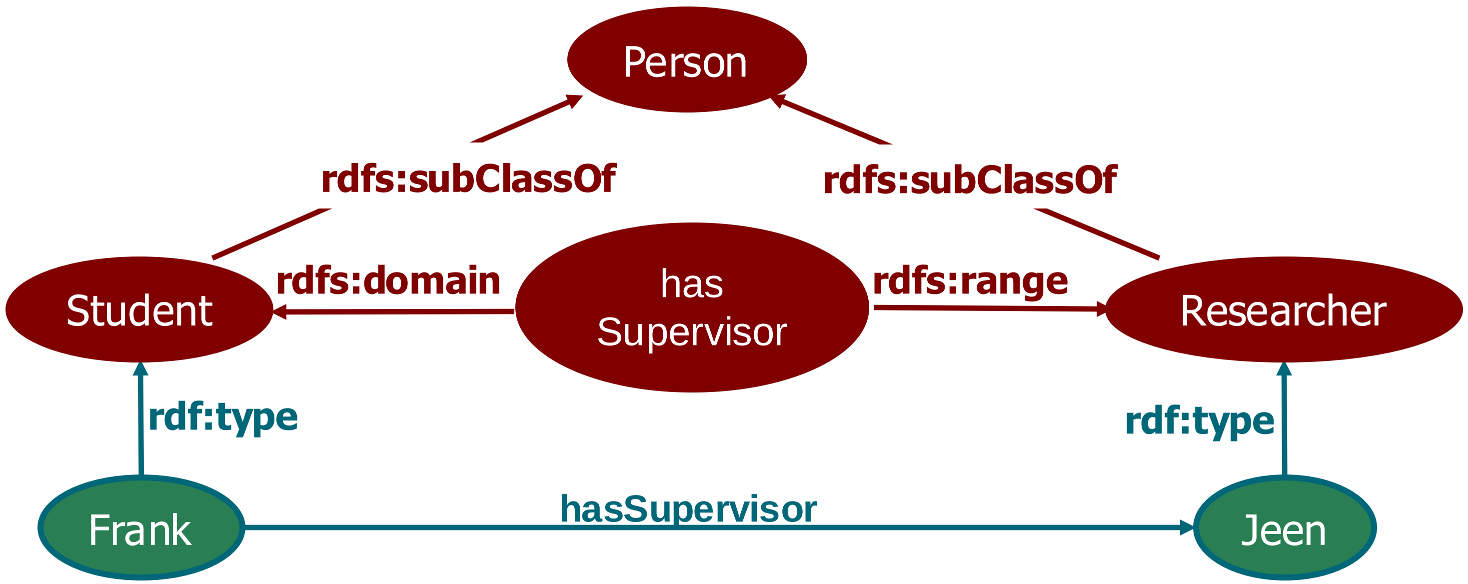
```
Frank rdf:type Student.
Jeen rdf:type Researcher.
Frank hasSupervisor Jeen.
```

RDFS-based KG: example

```
Student rdfs:subClassOf Person.
Researcher rdfs:subClassOf Person.
hasSupervisor rdfs:range Researcher.
hasSupervisor rdfs:domain Student.
Frank rdf:type Student.
Jeen rdf:type Researcher.
Frank hasSupervisor Jeen.
```

RDFS-based KG: example

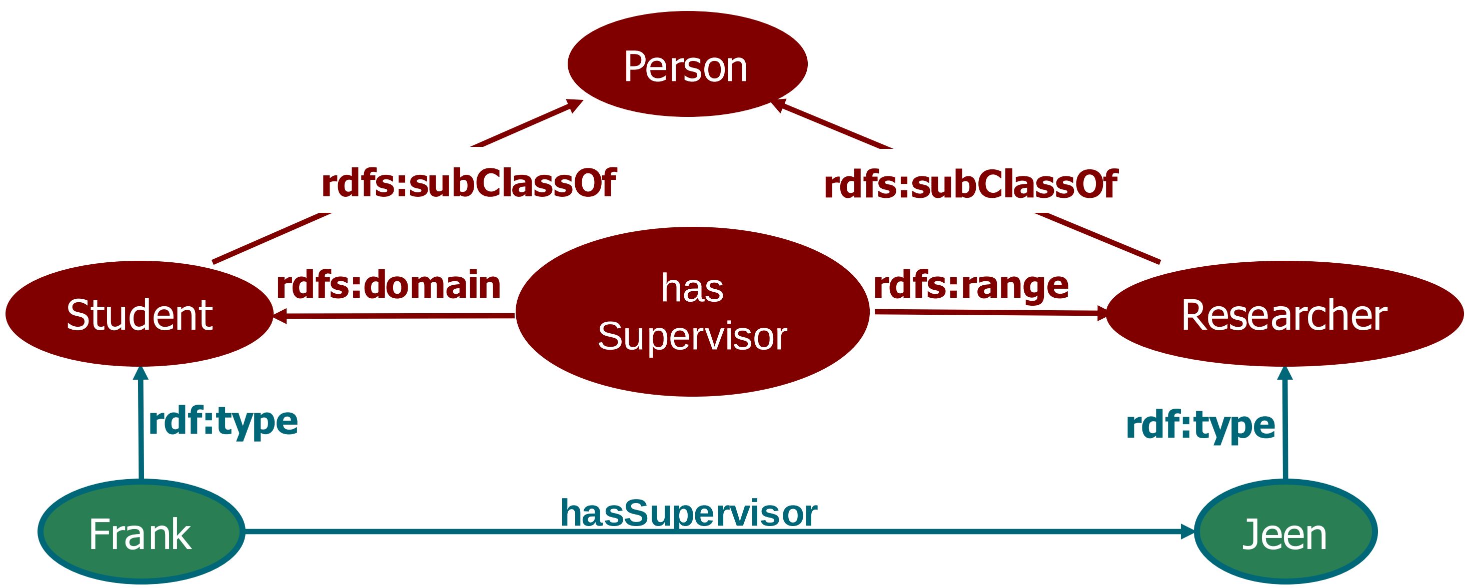
```
Student rdfs:subClassOf Person.
Researcher rdfs:subClassOf Person.
hasSupervisor rdfs:range Researcher.
hasSupervisor rdfs:domain Student.
Frank rdf:type Student.
Jeen rdf:type Researcher.
Frank hasSupervisor Jeen.
```



RDFS-based KG: is it a database?

Can this graph be regarded as a database?

```
Student rdfs:subClassOf Person.
Researcher rdfs:subClassOf Person.
hasSupervisor rdfs:range Researcher.
hasSupervisor rdfs:domain Student.
Frank rdf:type Student.
Jeen rdf:type Researcher.
Frank hasSupervisor Jeen.
```

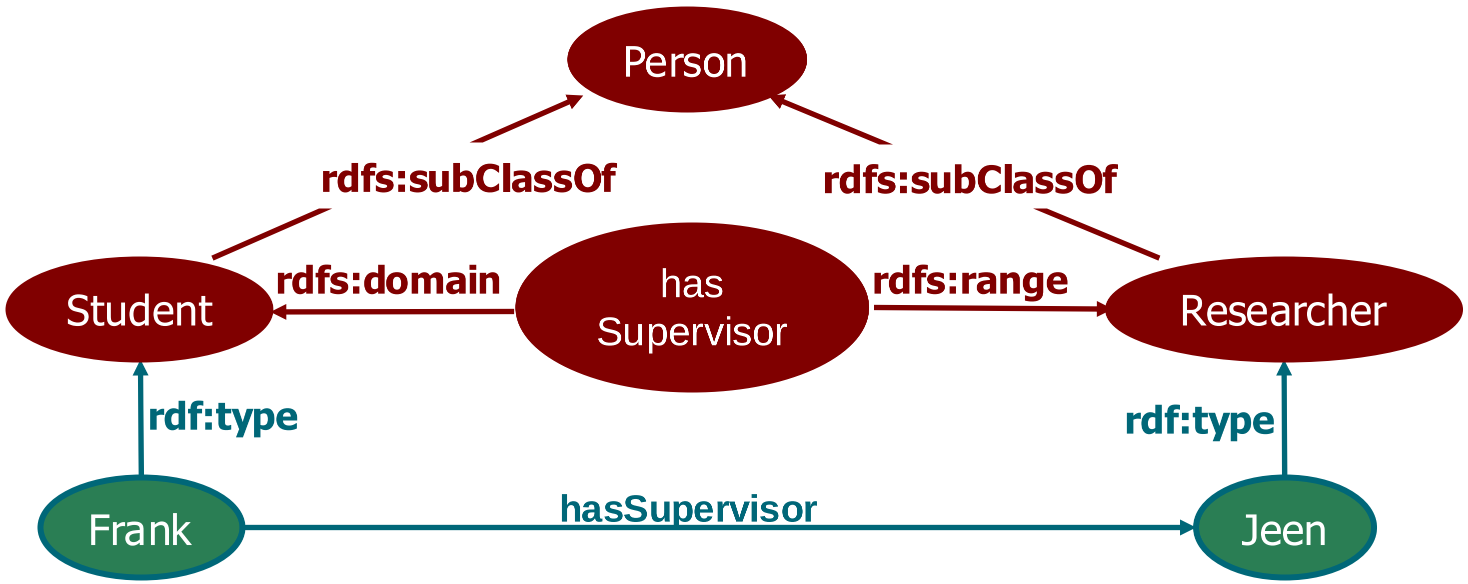


RDFS-based KG: is it a database?

This graph cannot be regarded as a database:

- In a database, a fact that is not present is considered false (Closed World Assumption); it follows that Frank is NOT a person if we regard the graph as a database
- So Frank is a student but NOT a person, incoherently with (Student rdfs:subClassOf Person)
- Similarly, Jeen is a supervisor of Frank, but is not a researcher, incoherently with (has Supervisor rdfs:range Researcher)

```
Student rdfs:subClassOf Person.
Researcher rdfs:subClassOf Person.
hasSupervisor rdfs:range Researcher.
hasSupervisor rdfs:domain Student.
Frank rdf:type Student.
Jeen rdf:type Researcher.
Frank hasSupervisor Jeen.
```

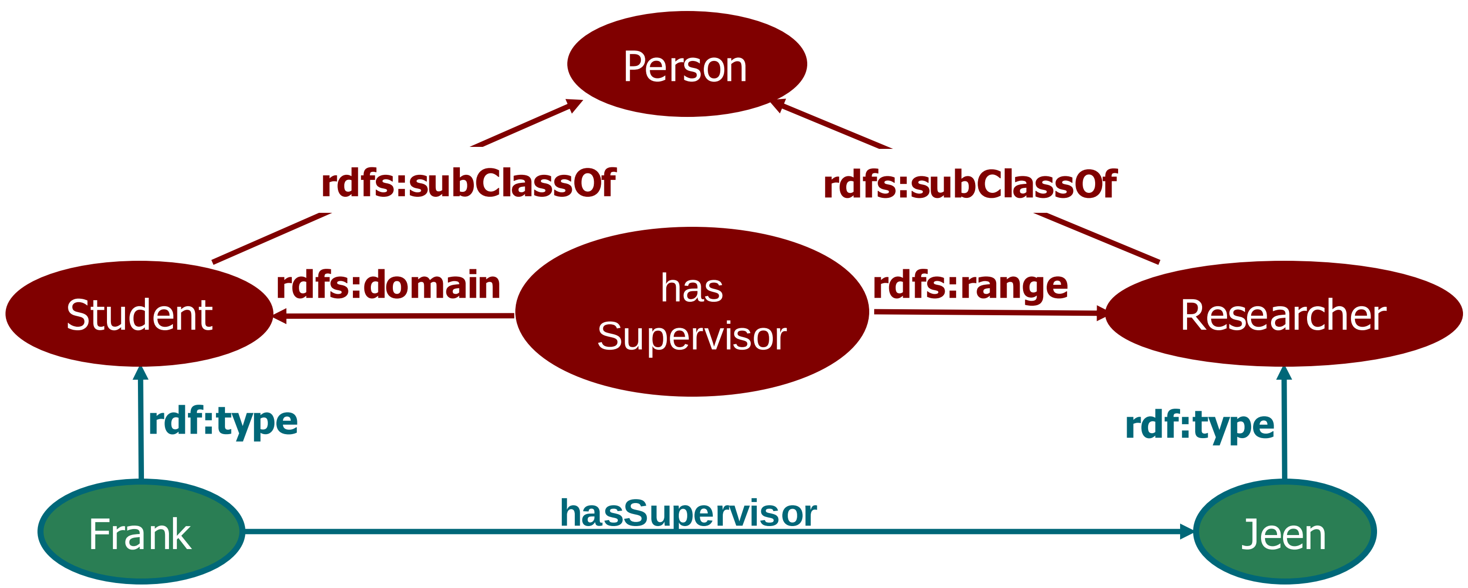


RDFS-based KG: is it a database?

This graph cannot be regarded as a database:

- Also, if we regard the graph as a database, we would conclude that Frank is not a researcher (because of the Closed World Assumption), although there is nothing in the knowledge that we have expressed that allows us to make such a conclusion

```
Student rdfs:subClassOf Person.
Researcher rdfs:subClassOf Person.
hasSupervisor rdfs:range Researcher.
hasSupervisor rdfs:domain Student.
Frank rdf:type Student.
Jeen rdf:type Researcher.
Frank hasSupervisor Jeen.
```



RDFS-based Knowledge Graph: semantics

- We have observed that considering a KG a database is problematic

- Indeed, we already have the notion of graph database, and therefore there are good reasons to consider KGs a different formalism with respect to a simple database

- So what is a KG?
  - We formalize a KG in logic, so that a KG will be seen as a logical theory
  - In order to do that, we have to specify the semantics of KG

RDFS-based Knowledge Graph: semantics

The semantics is based on the notion of interpretation. An interpretation $\boldsymbol{I}$ for a KG K is the formal representation of one world among all the possibile worlds shaped according to the alphabet and the structure of K.

More precisely, an interpretation $\boldsymbol{I}$ for a KG K is constituted by:

- the interpretation domain $\boldsymbol{\Delta^I}$ of $\boldsymbol{I}$, that is simply the set of all possible URIs;
- the intepretation function for $\boldsymbol{I} = <\boldsymbol{I_0}, \boldsymbol{I_1}, \boldsymbol{I_2}>$, that assigns
  - itself to every URI d, i.e., $\boldsymbol{I_0}(\mathrm{d}) = \mathrm{d}$ for every URI d;
  - an element $\boldsymbol{I_0}(\mathrm{d})$ in $\boldsymbol{\Delta^I}$ to every blank node d in K;
  - a subset $\boldsymbol{I_1}(\mathrm{C})$ of $\boldsymbol{\Delta^I}$ to each element C playing the role of class in K;
  - a subset $\boldsymbol{I_2}(\mathrm{P})$ of $\boldsymbol{\Delta^I} \times \boldsymbol{\Delta^I}$ to each element P playing the role of a property in K;

such that a set of predefined conditions are satisfied (see later).

In other words, every URI is interpreted in $\boldsymbol{I}$ as itself, each blank node is interpreted in $\boldsymbol{I}$ as an object in URI, every class is interpreted in $\boldsymbol{I}$ as a subset of the interpretation domain, and each property is interpreted in $\boldsymbol{I}$ as a binary relation over the interpretation domain. The predefined conditions determine how to interpret the predefined symbols of the alphabet (the built-in predicates **`rdfs:Resource, rdfs:Class, rdf:type, rdfs:subClassOf, rdf:Property, rdfs:subPropertyOf, rdfs:domain, rdfs:range`**) coherently with their intended meaning.

In what follows, $\boldsymbol{I_1}(\mathrm{X})$ and $\boldsymbol{I_2}(\mathrm{X})$ will be called the extension of X in $\boldsymbol{I}$.

RDFS-based Knowledge Graph: semantics

The predefined conditions that any interpretation $\boldsymbol{I} = \langle \boldsymbol{I_0}, \boldsymbol{I_1}, \boldsymbol{I_2} \rangle$ must satisfy determine the predefined extensions for all the built–in predicates, as follows:

- $\boldsymbol{I_1}$(`rdfs:Resource`) = $\boldsymbol{\Delta^I}$
- $\boldsymbol{I_1}$(`rdfs:Class`) ∋ `rdfs:Resource`, `rdfs:Class`, `rdfs:Property`
- $\boldsymbol{I_1}$(`rdfs:Property`) ∋ `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdf:Type`, `rdfs:range`, `rdfs:domain`
- $\boldsymbol{I_2}$(`rdf:type`) = { (a,b) | a ∈ $\boldsymbol{I_1}$(b) }
- $\boldsymbol{I_2}$(`rdfs:subClassOf`) = { (a,b) | a,b ∈ $\boldsymbol{I_1}$(`rdfs:Class`) and $\boldsymbol{I_1}$(a) ⊆ $\boldsymbol{I_1}$(b) }
- $\boldsymbol{I_2}$(`rdfs:subPropertyOf`) = { (a,b) | a,b ∈ $\boldsymbol{I_1}$(`rdfs:Property`) and $\boldsymbol{I_2}$(a) ⊆ $\boldsymbol{I_2}$(b) }
- $\boldsymbol{I_2}$(`rdfs:range`) = { (a,b) | a ∈ $\boldsymbol{I_1}$(`rdf:Property`), b ∈ $\boldsymbol{I_1}$(`rdfs:Class`) and { c | ∃ d . (d,c) ∈ $\boldsymbol{I_2}$(a) } ⊆ $\boldsymbol{I_1}$(b) }
- $\boldsymbol{I_2}$(`rdfs:domain`) = { (a,b) | a ∈ $\boldsymbol{I_1}$(`rdf:Property`), b ∈ $\boldsymbol{I_1}$(`rdfs:Class`) and { c | ∃ d . (c,d) ∈ $\boldsymbol{I_2}$(a) } ⊆ $\boldsymbol{I_1}$(b) }

Interpretation of an RDFS-based KG: example

KG K:

`Student` **`rdfs:subClassOf`** `Person`

`Researcher` **`rdfs:subClassOf`** `Person`

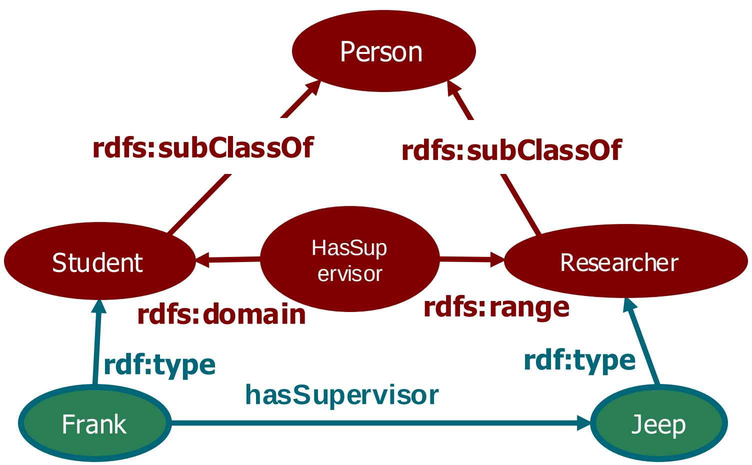
`hasSupervisor` **`rdfs:range`** `Researcher`

`hasSupervisor` **`rdfs:domain`** `Student`

`Frank` **`rdf:type`** `Student`

`Jeen` **`rdf:type`** `Researcher`

`Frank` **`hasSupervisor`** `Jeen`



Interpretation I for K (I_0(d)=d for every URI d)

I_1(Student) = { Frank,Paul }

I_1(Person) = { Jeen,Bob }

I_1(Researcher) = { }

I_2(hasSupervisor) = { (Frank,Jeen) }

I_1(**rdfs:Resource)={Student,Person,Paul,Researcher,Bob,Frank,Jeen, hasSupervisor,rdfs:Class,rdfs:Resource,rdf:Property,rdf:type, rdfs:subClassOf,rdfs:subPropertyOf,rdfs:range,rdfs:domain}**

I_1(**rdfs:Class)={rdfs:Class,rdfs:Resource,rdf:Property,Person, Researcher,Student}**

I_1(**rdf:Property)={hasSupervisor,rdf:type,rdfs:subClassOf, rdfs:subPropertyOf,rdfs:range,rdfs:domain}**

I_2(**rdf:type)={(Frank,Student},(Paul,Student),(Jeen,Person), (Bob,Person),(Student,rdfs:Class),(Person,rdfs:Class), (Researcher,rdfs:Class),(rdfs:Resource,rdfs:Class), (rdf:Property,rdfs:Class),(rdfs:Class,rdfs:Class), (rdf:type,rdf:Property),(HasSupervisor,rdf:Property),**

I_2(**rdfs:subClassOf)={(rdfs:Class,rdfs:Resource), (rdfs:Class,rdfs:Class),(rdfs:Resource,rdfs:Resource), (rdf:Property,rdfs:Resource),(rdf:Property,rdf:Property), (Student,Student),(Person,Person),(Researcher,Researcher), (Researcher,Person)}**

I_1(**rdfs:subPropertyOf)={ }**

I_1(**rdfs:range)={(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdf:type,rdfs:Class), (rdfs:range,rdfs:Class),(rdfs:domain,rdfs:Class)}**

I_1(**rdfs:domain)={(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class)}**

Interpretation of an RDFS-based KG: example

KG K:

`Student` **`rdfs:subClassOf`** `Person`

`Researcher` **`rdfs:subClassOf`** `Person`

`hasSupervisor` **`rdfs:range`** `Researcher`

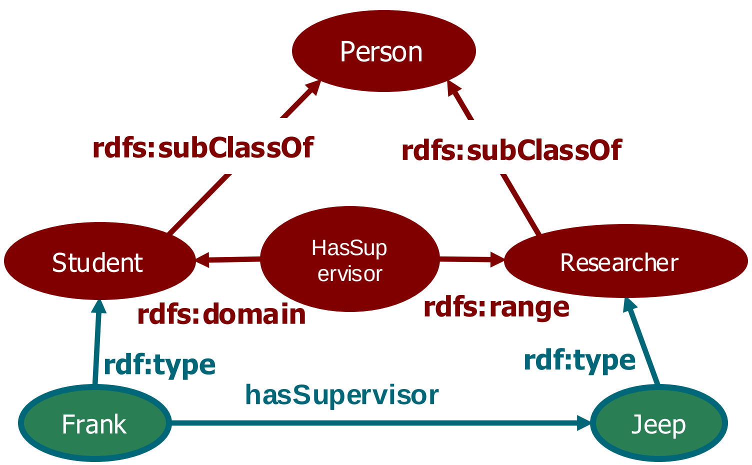
`hasSupervisor` **`rdfs:domain`** `Student`

`Frank` **`rdf:type`** `Student`

`Jeen` **`rdf:type`** `Researcher`

`Frank` **`hasSupervisor`** `Jeen`

Is ***I*** a "good" interpretation for K?



Interpretation I for K (I_0(d)=d for every URI d)

I_1(Student) = { Frank,Paul }

I_1(Person) = { Jeen,Bob }

I_1(Researcher) = { }

I_2(hasSupervisor) = { (Frank,Jeen) }

I_1(rdfs:Resource)={Student,Person,Paul,Researcher,Bob,Frank,Jeen, hasSupervisor,rdfs:Class,rdfs:Resource,rdf:Property,rdf:type, rdfs:subClassOf,rdfs:subPropertyOf,rdfs:range,rdfs:domain}

I_1(rdfs:Class)={rdfs:Class,rdfs:Resource,rdf:Property,Person, Researcher,Student}

I_1(rdf:Property)={hasSupervisor, rdf:type,rdfs:subClassOf, rdfs:subPropertyOf,rdfs:range,rdfs:domain}

I_1(rdf:type)={(Frank,Student},(Paul,Student),(Jeen,Person), (Bob,Person),(Student,rdfs:Class),(Person,rdfs:Class), (Researcher,rdfs:Class),(rdfs:Resource,rdfs:Class), (rdf:Property,rdfs:Class),(rdfs:Class,rdfs:Class), (rdf:type,rdf:Property),(HasSupervisor,rdf:Property)}

I_1(rdfs:subClassOf)={(rdfs:Class,rdfs:Resource), (rdfs:Class,rdfs:Class),(rdfs:Resource,rdfs:Resource), (rdf:Property,rdfs:Resource),(rdf:Property,rdf:Property), (Student,Student),(Person,Person),(Researcher,Researcher), (Researcher,Person)}

I_2(rdfs:subPropertyOf)={ }

I_2(rdfs:range)={(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdf:type,rdfs:Class), (rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class), }

I_2(rdfs:domain)={(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class)}

RDFS-based Knowledge Graph: semantics

BASIC QUESTION: When is the world represented by an interpretation ***I*** for K coherent with the conditions expressed in K? I.e., when is ***I*** good for K, or when is ***I*** a model of K? The answer is exactly in the definition of "model".

A model of KG K is an interpretation $I = <I_1, I_2>$ for K such that:

- for every element c playing the role of a class in K: c ∈ I_1(**`rdfs:Class`**)
- for every element p playing the role of a property in K: p ∈ I_1(**`rdf:Property`**)
- every triple in K is satisfied by ***I***, where the triple (X P Y) is satisfied by ***I*** if the pair (I_0(X),I_0(Y)) is in the extension of P in ***I***, i.e. if (I_0(X),I_0(Y)) ∈ I_2(P)

While an interpretation ***I*** for a KG K is the representation of one of the possible worlds coherent with the alphabet and the structure of K, a model is the representation of one of the possible worlds that are coherent with the alphabet of K, the structure of K and the knowledge expressed by (the triples in) K.

Relationship with formal logic

The definition of the semantics of RDFS KGs clarifies that there is an intuitive relationship with formal logic.

- Class → Unary Predicate (one argument)
- Property → Binary Predicate (two arguments)
- Here are the sentences in first-order logic that intuitively correspond to the triples of RDFS-based KGs (a and b are constants):
  - (a **`rdf:type`** C) — $C(a)$
  - (C **`rdfs:subClassOf`** D) — $\forall x\ C(x) \rightarrow D(x)$
  - (R **`rdfs:domain`** C) — $\forall x \forall y\ R(x,y) \rightarrow C(x)$
  - (R **`rdfs:range`** C) — $\forall x \forall y\ R(x,y) \rightarrow C(y)$
  - (R **`rdfs:subPropertyOf`** **Q**) — $\forall x \forall y\ R(x,y) \rightarrow Q(x,y)$
  - (a R b) — $R(a,b)$

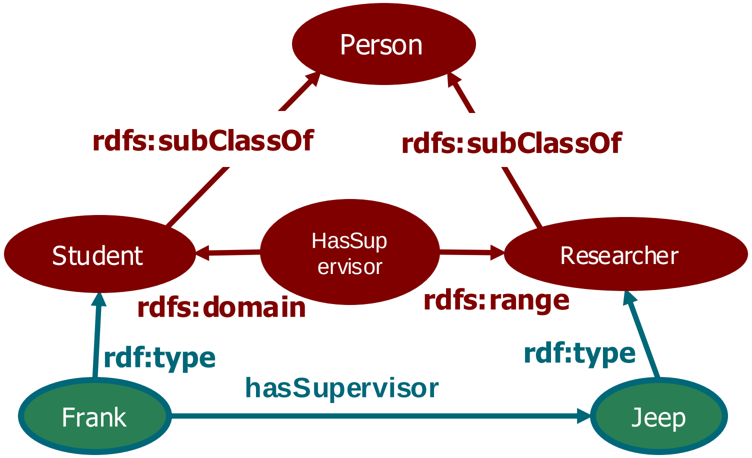
Interpretation of an RDFS-based KG: example

KG K:

```
Student rdfs:subClassOf Person
Researcher rdfs:subClassOf Person
hasSupervisor rdfs:range Researcher
hasSupervisor rdfs:domain Student
Frank rdf:type Student
Jeen rdf:type Researcher
Frank hasSupervisor Jeen
```

Note that I is not a model of K, because not all the knowledge represented in K is reflected by I:

- the first triple of K imposes that the extension of Student is a subset of the extension of Person, but I_1(Student) $\nsubseteq$ I_1(Person)
- the third triple of K imposes that the range of hasSupervisor is Researcher, but Jeen is not in I_1(Researcher)



Interpretation I for K (I_0(d)=d for every URI d)

I_1(Student) = { Frank,Paul }

I_1(Person) = { Jeen,Bob }

I_1(Researcher) = { }

I_2(hasSupervisor) = { (Frank,Jeen) }

I_1(rdfs:Resource)={Student,Person,Paul,Researcher,Bob,Frank,Jeen, hasSupervisor,rdfs:Class,rdfs:Resource,rdf:Property,rdf:type, rdfs:subClassOf,rdfs:subPropertyOf,rdfs:range,rdfs:domain}

I_1(rdfs:Class)={rdfs:Class,rdfs:Resource,rdf:Property,Person, Researcher,Student}

I_1(rdf:Property)={hasSupervisor, rdf:type,rdfs:subClassOf, rdfs:subPropertyOf,rdfs:range,rdfs:domain}

I_1(rdf:type)={(Frank,Student},(Paul,Student),(Jeen,Person), (Bob,Person),(Student,rdfs:Class),(Person,rdfs:Class), (Researcher,rdfs:Class),(rdfs:Resource,rdfs:Class), (rdf:Property,rdfs:Class),(rdfs:Class,rdfs:Class), (rdf:type,rdf:Property),(HasSupervisor,rdf:Property)}

I_1(rdfs:subClassOf)={(rdfs:Class,rdfs:Resource), (rdfs:Class,rdfs:Class),(rdfs:Resource,rdfs:Resource), (rdf:Property,rdfs:Resource),(rdf:Property,rdf:Property), (Student,Student),(Person,Person),(Researcher,Researcher), (Researcher,Person)}

I_1(rdfs:subPropertyOf)={ }

I_1(rdfs:range)={(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdf:type,rdfs:Class), (rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class), }

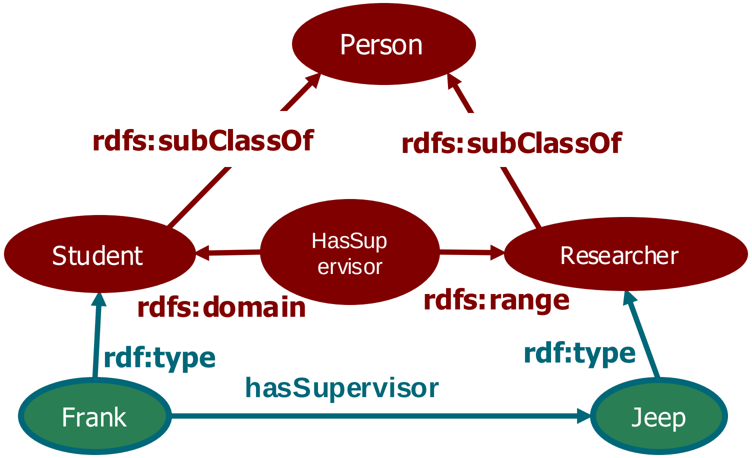
I_1(rdfs:domain)={(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class)}

Interpretation of an RDFS-based KG: example

KG K:

```
Student rdfs:subClassOf Person
Researcher rdfs:subClassOf Person
hasSupervisor rdfs:range Researcher
hasSupervisor rdfs:domain Student
Frank rdf:type Student
Jeen rdf:type Researcher
Frank hasSupervisor Jeen
```

Note that ***I*** is now a model of K, because all the knowledge represented in K is reflected in the "world" represented by ***I***



Interpretation I for K (I_0(d)=d for every URI d)

I_1(Student) = { Frank,Paul }

I_1(Person) = { Jeen,Bob,Frank,Paul }

I_1(Researcher) = { Jeen }

I_2(hasSupervisor) = { (Frank,Jeen) }

I_1(rdfs:Resource)={Student,Person,Paul,Researcher,Bob,Frank,Jeen, hasSupervisor,rdfs:Class,rdfs:Resource,rdf:Property,rdf:type, rdfs:subClassOf,rdfs:subPropertyOf,rdfs:range,rdfs:domain}

I_1(rdfs:Class)={rdfs:Class,rdfs:Resource,rdf:Property,Person, Researcher,Student}

I_1(rdf:Property)={hasSupervisor, rdf:type,rdfs:subClassOf, rdfs:subPropertyOf,rdfs:range,rdfs:domain}

I_1(rdf:type) = {(Frank,Student},(Paul,Student),(Jeen,Person), (Jeen,Researcher),(Bob,Person),(Student,rdfs:Class), (Person,rdfs:Class),(Researcher,rdfs:Class), (rdfs:Resource,rdfs:Class),(rdf:Property,rdfs:Class), (rdfs:Class,rdfs:Class),(rdf:type,rdf:Property), (HasSupervisor,rdf:Property}

I_1(rdfs:subClassOf)={(rdfs:Class,rdfs:Resource), (rdfs:Class,rdfs:Class),(rdfs:Resource,rdfs:Resource), (rdf:Property,rdfs:Resource),(rdf:Property,rdf:Property), (Student,Student),(Person,Person),(Researcher,Researcher), (Researcher,Person)}

I_1(rdfs:subPropertyOf) = { }

I_1(rdfs:range) = {(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdf:type,rdfs:Class), (rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class), }

I_1(rdfs:domain) = {(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class)}

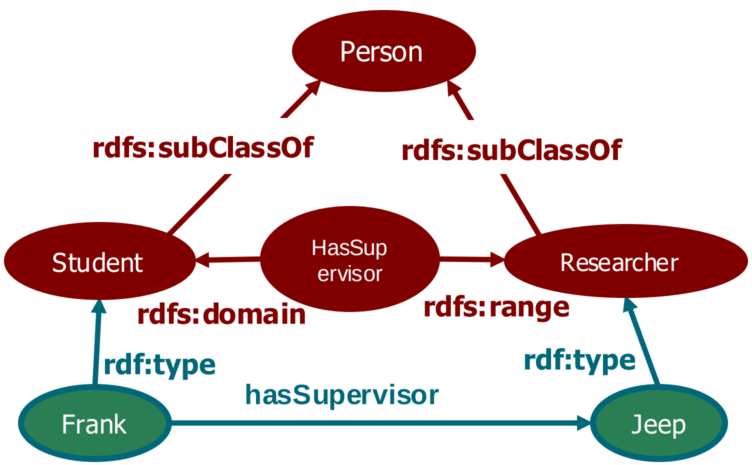
Interpretation of an RDFS-based KG: example

KG K:

```
Student rdfs:subClassOf Person
Researcher rdfs:subClassOf Person
hasSupervisor rdfs:range Researcher
hasSupervisor rdfs:domain Student
Frank rdf:type Student
Jeen rdf:type Researcher
Frank hasSupervisor Jeen
```

J differs from the previous model *I* because it adds some knowledge an Alice and Bob.

Is *J* a model of K?



Interpretation J for K (I_0(d)=d for every URI d)

J_1(Student) = { Frank,Paul,**Alice** }

J_1(Person) = { Jeen,Bob,Frank,Paul,**Alice**,**Bob** }

J_1(Researcher) = { Jeen,**Bob** }

J_2(hasSupervisor) = { (Frank,Jeen),**(Alice,Bob)** }

J_1(rdfs:Resource)={Student,Person,Paul,Researcher,Bob,Frank,Jeen, Alice,hasSupervisor,rdfs:Class,rdfs:Resource,rdf:Property, rdf:type,rdfs:subClassOf,rdfs:subPropertyOf,rdfs:range, rdfs:domain}

J_1(rdfs:Class)={rdfs:Class,rdfs:Resource,rdf:Property,Person, Researcher,Student}

J_1(rdf:Property)={hasSupervisor, rdf:type,rdfs:subClassOf, rdfs:subPropertyOf,rdfs:range,rdfs:domain}

J_1(rdf:type)={(Frank,Student},(Paul,Student),(Jeen,Person), (Jeen,Researcher),(Bob,Person),(Student,rdfs:Class), (Person,rdfs:Class),(Researcher,rdfs:Class), (rdfs:Resource,rdfs:Class),(rdf:Property,rdfs:Class), (rdfs:Class,rdfs:Class),(rdf:type,rdf:Property), (HasSupervisor,rdf:Property),(Alice,Student},(Alice,Person), (Bob,Person),(Bob,Researcher)}

J_1(rdfs:subClassOf)={(rdfs:Class,rdfs:Resource), (rdfs:Class,rdfs:Class),(rdfs:Resource,rdfs:Resource), (rdf:Property,rdfs:Resource),(rdf:Property,rdf:Property), (Student,Student),(Person,Person),(Researcher,Researcher), (Researcher,Person)}

J_1(rdfs:subPropertyOf) = { }

J_1(rdfs:range) = {(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdf:type,rdfs:Class), (rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class), }

J_1(rdfs:domain) = {(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class)}

Interpretation of an RDFS-based KG: example

KG K:

Student **rdfs:subClassOf** Person

Researcher **rdfs:subClassOf** Person

hasSupervisor **rdfs:range** Researcher

hasSupervisor **rdfs:domain** Student

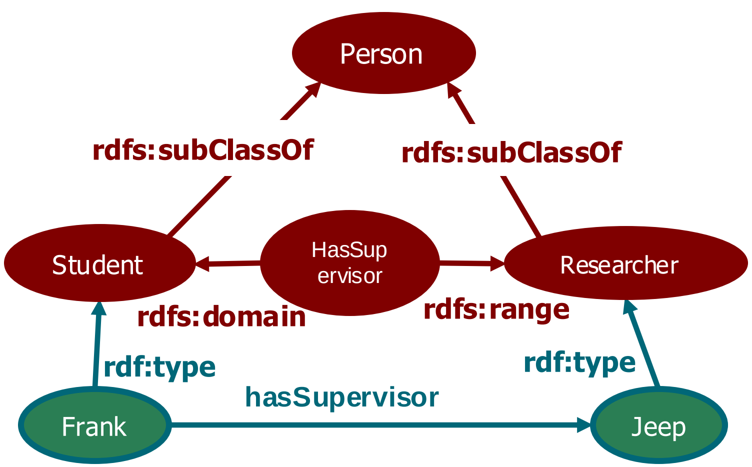
Frank **rdf:type** Student

Jeen **rdf:type** Researcher

Frank **hasSupervisor** Jeen

Is ***J*** a model of K? Yes, because

- all the knowledge of K is reflected by ***J***, and
- the fact hat has ***J*** more knowledge wrt K is allowed, because we cannot assume that facts that are not represented by K are false (Open World Assumption, see later).



Interpretation J for K (I_0(d)=d for every URI d)

J_1(Student) = { Frank,Paul,**Alice** }

J_1(Person) = { Jeen,Bob,Frank,Paul,**Alice,Bob** }

J_1(Researcher) = { Jeen,**Bob** }

J_2(hasSupervisor) = { (Frank,Jeen),**(Alice,Bob)** }

J_1(rdfs:Resource)={Student,Person,Paul,Researcher,Bob,Frank,Jeen, Alice,hasSupervisor,rdfs:Class,rdfs:Resource,rdf:Property, rdf:type,rdfs:subClassOf,rdfs:subPropertyOf,rdfs:range, rdfs:domain}

J_1(rdfs:Class)={rdfs:Class,rdfs:Resource,rdf:Property,Person, Researcher,Student}

J_1(rdf:Property)={hasSupervisor, rdf:type,rdfs:subClassOf, rdfs:subPropertyOf,rdfs:range,rdfs:domain}

J_1(rdf:type)={(Frank,Student},(Paul,Student),(Jeen,Person), (Jeen,Researcher),(Bob,Person),(Student,rdfs:Class), (Person,rdfs:Class),(Researcher,rdfs:Class), (rdfs:Resource,rdfs:Class),(rdf:Property,rdfs:Class), (rdfs:Class,rdfs:Class),(rdf:type,rdf:Property), (HasSupervisor,rdf:Property),(Alice,Student},(Alice,Person), (Bob,Person),(Bob,Researcher)}

J_1(rdfs:subClassOf)={(rdfs:Class,rdfs:Resource), (rdfs:Class,rdfs:Class),(rdfs:Resource,rdfs:Resource), (rdf:Property,rdfs:Resource),(rdf:Property,rdf:Property), (Student,Student),(Person,Person),(Researcher,Researcher), (Researcher,Person)}

J_1(rdfs:subPropertyOf) = { }

J_1(rdfs:range) = {(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdf:type,rdfs:Class), (rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class), }

J_1(rdfs:domain) = {(rdfs:subClassOf,rdfs:Class), (rdfs:subPropertyOf,rdf:Property),(rdfs:range,rdfs:Class), (rdfs:domain,rdfs:Class)}

Exercise

Write (in any format you like) an RDFS-based KG representing the properties of a single oganization in terms of the following statements regarding the URIs :Employee, :Office, :worksIn, :isHeadOf, :John, :Mary, :SalesOffice. :Unit.

1. Those who work in offices are employees;
2. If an employee works in the organization, then (s)he works in an office;
3. An employee who is the head of an office works in that office;
4. John and Mary are employees;
5. Sale offices are particular offices;
6. John works in a sale office;
7. Mary is the head of a sale office;
8. Offices are particular organization units
9. Laboratories are also particular organization units

Solution

:Employee rdf:type rdfs:Class .
:Office rdf:type rdfs:Class .
:worksIn rdf:type rdf:Property .
:isHeadOf rdf:type rdf:Property .

1. Those who work in offices are employees;
2. If an employee works in the organization, then (s)he works in an office;
3. An employee who is the head of an office works in that office;
4. John and Mary are employees;
5. Sale offices are particular offices;
6. John works in a sale office;
7. Mary is a the head of a sale office;
8. Offices are particular organization units
9. Laboratories are also particular organization units

1. :worksIn rdfs:domain :Employee .

2. :worksIn rdfs:range :Office .

3. :isHeadOf rdfs:subPropertyOf :worksIn .

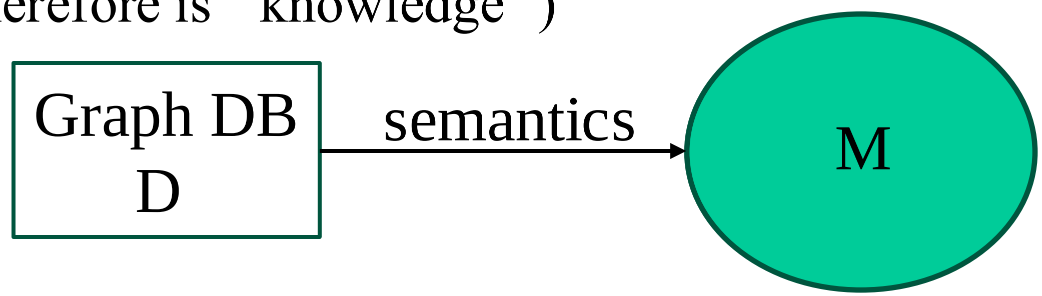
4. :John rdf:type :Employee .
4. :Mary rdf:type :Employee .
5. :SalesOffice rdfs:subClassOf :Office .

6. :John :works :SalesOffice.
7. :Mary :isHeadOf :SalesOffice .
8 :Office rdfs:subClassOf Unit.

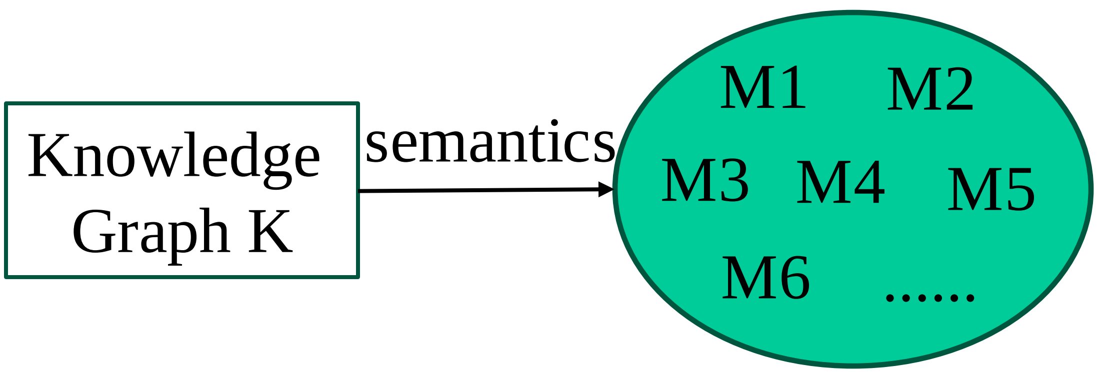
9. :Laboratory rdfs:subClassOf Unit.

Summarizing, differently from a DB, a KG has many models

- Closed world assumption: what is not represented in the database (graph) is false
- Open world assumption: we cannot assume that what is not represented in the database (graph) is false and therefore there are many ways to “add” information to the graph without becomeing incoherent with the graph itself. And some information that we can add is not only “possible”, but “necessary” (and therefore is “knowledge”)



A database (relational or graph-oriented), by virtue of the Closed World Assumption, corresponds to one model of a logical theory!



A KG, by virtue of the Open World Assumption, is a theory with **many models**

The notion of logical implication

Consider the following KG K:

P **`rdfs:subPropertyOf`** Q.
Q **`rdfs:subPropertyOf`** R.
R **`rdfs:range`** C.
a P b.

Question 1: is there a model $\boldsymbol{I}$ of K where $(b,a) \in \boldsymbol{I}_2(R)$?

Question 2: is there a model $\boldsymbol{I}$ of K where $(b,a) \notin \boldsymbol{I}_2(R)$?

The notion of logical implication

Consider the following KG K:

P **`rdfs:subPropertyOf`** Q.
Q **`rdfs:subPropertyOf`** R.
R **`rdfs:range`** C.
a P b.

Question 1: is there a model ***I*** of K where (b,a) $\in \boldsymbol{I_2}$(R)?

Answer 1: YES

Question 2: is there a model ***I*** of K where (b,a) $\notin \boldsymbol{I_2}$(R)?

Answer 2: YES

The notion of logical implication

Consider the following KG K:

P **`rdfs:subPropertyOf`** Q.
Q **`rdfs:subPropertyOf`** R.
R **`rdfs:range`** C.
a P b.

Question 1: is there a model $\boldsymbol{I}$ of K where (b,a) $\in \boldsymbol{I_2}$(R)?

Answer 1: YES

Question 2: is there a model $\boldsymbol{I}$ of K where (b,a) $\notin \boldsymbol{I_2}$(R)?

Answer 2: YES

CONCLUSION: (b,a) $\in$ R is possible but not necessary, given K

The notion of logical implication

Consider the following KG K:

P **`rdfs:subPropertyOf`** Q.
Q **`rdfs:subPropertyOf`** R.
R **`rdfs:range`** C.
a P b.

Question 1: is there a model $\boldsymbol{I}$ of K where $(a,b) \notin \boldsymbol{I_2}(R)$?

Question 2: is there a model $\boldsymbol{I}$ of K where $b \notin \boldsymbol{I_1}(C)$?

The notion of logical implication

Consider the following KG K:

P **`rdfs:subPropertyOf`** Q.
Q **`rdfs:subPropertyOf`** R.
R **`rdfs:range`** C.
a P b.

Question 1: is there a model $\boldsymbol{I}$ of K where $(a,b) \notin \boldsymbol{I_2}(R)$?

Answer: NO

Question 2: is there a model $\boldsymbol{I}$ of K where $b \notin \boldsymbol{I_1}(C)$?

Answer: NO

The notion of logical implication

Consider the following KG K:

P **`rdfs:subPropertyOf`** Q.
Q **`rdfs:subPropertyOf`** R.
R **`rdfs:range`** C.
a P b.

Question 1: is there a model I of K where $(a,b) \notin I_2(R)$? NO

because every model of K satisfies (P **`rdfs:subPropertyOf`** R)

Question 2: is there a model I of K where $b \notin I_1(C)$? NO

because every model of K satisfies (P **`rdfs:range`** C)

CONCLUSION: both $(a,b) \in R$ and $b \in C$ are necessarily true, given K

The notion of logical implication

In the above example, (P **`rdfs:subPropertyOf`** R), (P **`rdfs:range`** C), (a R b), and (b **`rdfs:type`** C) are necessarily true, given the knowledge represented in the graph K. To formalize this notion, we introduce the concept of logical implication.

We say that a triple T is logically implied by K (or, K logically implies T, written K $\models$ T, or T is inferred from K by deduction) if every model of K satisfies T.

In other words, the triples that are logically implied by K are those necessarily true in every model of K. In practice, if K constitutes the explicit knowledge that we have over a certain domain, the triples that are logically implied by K and do not appear in K constitute the knowledge that is implicitly valid over the domain, although we did not express it in the graph.

The notion of logical implication

Problem

Given K, can we compute all the triples that are logically implied by K?

We will address this problem by first focusing on instance-based triples, i.e., triples of the forms (X **`rdf:type`** Y) or (X P Y)

The notion of logical implication

Computing the Instance-based Completion of K, denoted *IBC*(K):

$K_1 \leftarrow K \cup \{$ T | the instance-based triple T is deducible from the predefined conditions on the built-in predicates (e.g., **`rdf:type rdf:type rdfs:Property`**) $\}$

repeat

$K_0 \leftarrow K_1$;

if $\exists$X,C,D such that (X **`rdf:type`** C), (C **`rdfs:subClassOf`** D) $\in K_1$ and (X **`rdf:type`** D) $\notin K_1$

then add (X **`rdf:type`** D) to K_1;

if $\exists$X,P,Y,Q such that (X P Y), (P **`rdfs:subPropertyOf`** Q) $\in K_1$ and (X Q Y) $\notin K_1$

then add (X Q Y) to K_1;

if $\exists$X,P,Y,Q such that (X P Y), (P **`rdfs:domain`** C) $\in K_1$ and (X **`rdf:type`** C) $\notin K_1$

then add (X **`rdf:type`** C) to K_1

if $\exists$X,P,Y,Q such that (X P Y), (P **`rdfs:range`** C) $\in K_1$ and (Y **`rdf:type`** C) $\notin K_1$

then add (Y **`rdf:type`** C) to K_1

until $K_1 = K_0$

The notion of logical implication

Theorem For every K, the procedure that computes *IBC*(K) terminates.
Proof. It is sufficient to observe that only a finite number of triples can be formed using the symbols in K. This means that after a finite number of the "repeat loop", we cannot add any more triple, and we will have $K_1 = K_0$, which is the condition for exiting the loop.

It is immediate to observe that, for every K, *IBC*(K) can be seen as an interpretation $\boldsymbol{I}^{IBC(K)}$ of K, simply by assigning through $\boldsymbol{I}_0$ to each blank node in K any URI that does not appear in K (different URIs to different blank nodes) and defining:

- $\boldsymbol{I}_1^{IBC(K)}$(**`rdfs:Resource`**), $\boldsymbol{I}_1^{IBC(K)}$(**`rdfs:class`**) and $\boldsymbol{I}_1^{IBC(K)}$(**`rdf:Property`**) in the obvious way
- $\boldsymbol{I}_1^{IBC(K)}$ (C) = { X | (X **`rdf:type`** C) ∈ *IBC*(K) } for every user-defined class C,
- $\boldsymbol{I}_2^{IBC(K)}$(P) = { (X,Y) | (X P Y) ∈ *IBC*(K) } for every user-defined property P.

The notion of logical implication

Theorem For every K, $\boldsymbol{I}^{IBC(\mathrm{K})}$ is a model of K.

Proof. We show that every triple T of K is satisfied by $\boldsymbol{I}^{IBC(\mathrm{K})}$.

1. All triples of the form (X **`rdf:type`** Y) or (X Y Z) in K are also in *IBC*(K) and therefore they are satisfied by $\boldsymbol{I}^{IBC(\mathrm{K})}$ by construction of $\boldsymbol{I}^{IBC(\mathrm{K})}$.
2. Consider a triple T of the form (C **`rdfs:subClassOf`** D) in K: since the procedure for computing has terminated, this means that there is no (X **`rdf:type`** C) ∈ *IBC*(K) such that (X **`rdf:type`** D) ∉ *IBC*(K) (otherwise the condition for exiting from the repeat loop would be false). We conclude that (C **`rdfs:subClassOf`** D) in K is satisfied by $\boldsymbol{I}^{IBC(\mathrm{K})}$.
3. For triples of type (X **`rdfs:domain`** Y),(X **`rdfs:range`** Y) and (X **`rdfs:subPropertyOf`** Y), we proceed as in case 2.

The notion of logical implication

Theorem If T is an instance-based triple, i.e., a triple of the form (X **`rdf:type`** Y) or of the form (X Y Z), then K ⊨ T if and only if T ∈ *IBC*(K). In other words, *IBC*(K) = { instance-based triple T | K ⊨ T }.

Proof: To prove that if K ⊨ T for an instance-based triple T, then T ∈ *IBC*(K), observe that K ⊨ T means that every model of K satisfies T. But we have seen that $\boldsymbol{I}^{IBC(\mathrm{K})}$ is a model of K that reflects exactly *IBC*(K). Therefore, we have that $\boldsymbol{I}^{IBC(\mathrm{K})}$ satisfies T, and since T is an instance-based triple, by construction of $\boldsymbol{I}^{IBC(\mathrm{K})}$ we have that T ∈ *IBC*(K).

Proving that "if T ∈ *IBC*(K) then K ⊨ T" is more difficult. In order to do that, we can proceed by induction on the number of triples added during the execution of the procedure that computes *IBC*(K). We leave this as an exercise for the students.

The notion of logical implication

We now consider triples that are not instance-based. In the theorems below, K is an RDFS KG. We leave the corresponding proofs as an exercise for the students.

Theorem If T is a triple of the form (C **`rdfs:subClassOf`** D), then K ⊨ T if and only if (x **`rdf:type`** D) ∈ *IBC*(K'), where K' = K ∪ { (x **`rdf:type`** C) }, with x blank node not appearing in K.

Theorem If T is a triple of the form (P **`rdfs:subPropertyOf`** Q), then K ⊨ T if and only if (x Q y) ∈ *IBC*(K'), where K' = K ∪ { (x P y) }, with x,y blank nodes not appearing in K.

Theorem If T is a triple of the form (R **`rdfs:domain`** C), then K ⊨ T if and only if (x **`rdf:type`** C) ∈ *IBC*(K'), where K' = K ∪ { (x R y) }, with x,y blank nodes not appearing in K.

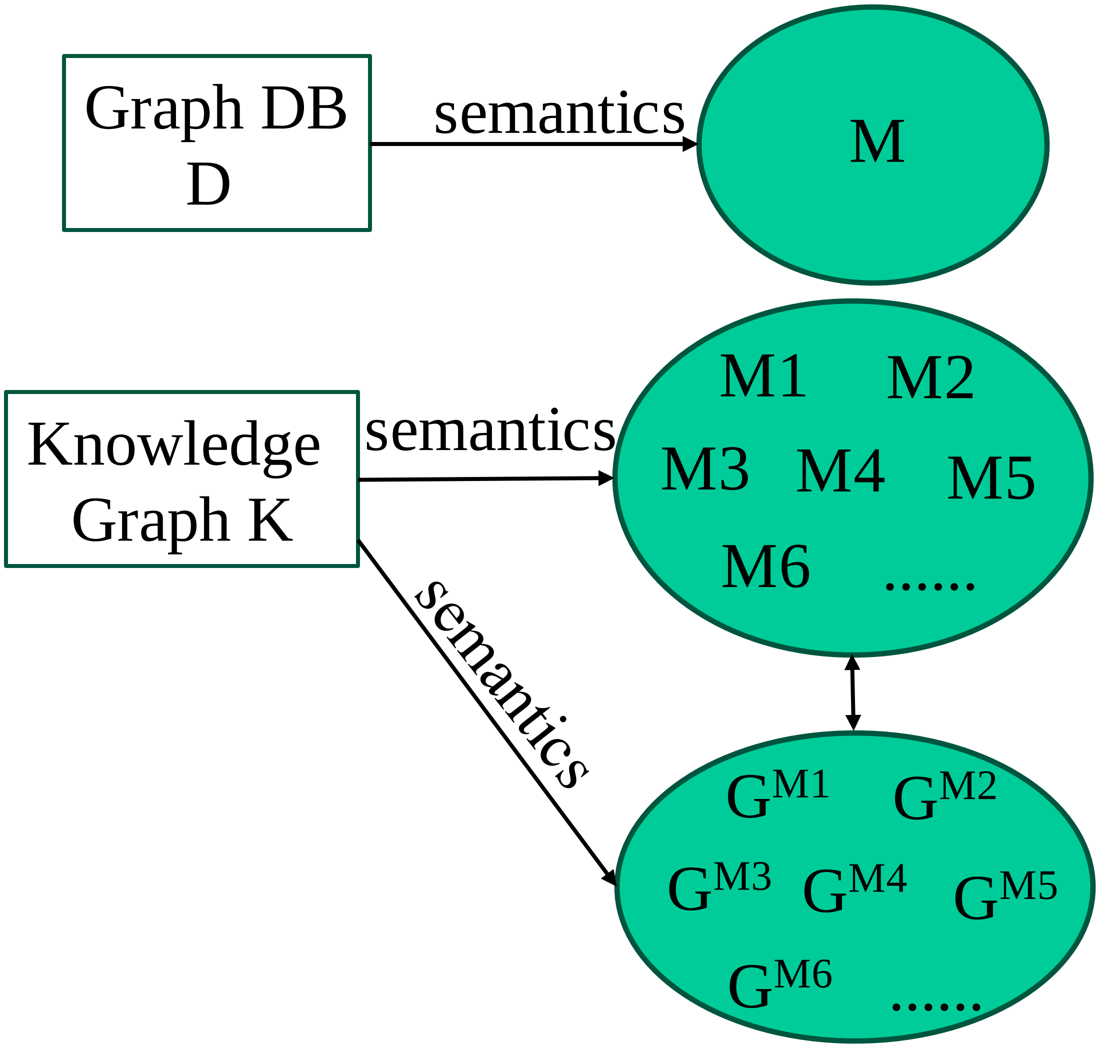
Theorem If T is a triple of the form (R **`rdfs:range`** C), then K ⊨ T if and only if (x **`rdf:type`** C) ∈ *IBC*(K'), where K' = K ∪ { (y R x) }, with x,y blank nodes not appearing in K.

Interpretations as Knowledge Graphs

Note that an interpretation $\boldsymbol{I}$ for a KG K can be seen as an RDFS-based Knowledge Graph $G^{\boldsymbol{I}}$, defined as follows:

- for every $x \in \boldsymbol{I}_1(C)$, $G^{\boldsymbol{I}}$ contains the triple (x **`rdf:type`** C)
- for every $(x,y) \in \boldsymbol{I}_2(R)$, $G^{\boldsymbol{I}}$ contains the triple (x R C)
- for every C,D such that $\boldsymbol{I}_1(C) \subseteq \boldsymbol{I}_1(D)$, $G^{\boldsymbol{I}}$ contains the triple (C **`rdfs:subClassOf`** D)
- for every P,Q such that $\boldsymbol{I}_2(P) \subseteq \boldsymbol{I}_2(Q)$, $G^{\boldsymbol{I}}$ contains the triple (P **`rdfs:subPropertyOf`** Q)
- for every P,C such that $\forall(w,z) \in \boldsymbol{I}_2(P)$: $w \in \boldsymbol{I}_1(C)$, $G^{\boldsymbol{I}}$ contains the triple (P **`rdfs:domain`** C)
- for every P,C such that $\forall(w,z) \in \boldsymbol{I}_2(P)$: $z \in \boldsymbol{I}_1(C)$, $G^{\boldsymbol{I}}$ contains the triple (P **`rdfs:range`** C)

Differently from a DB, a KG has many models: How does this impact on query evaluation?



Querying the graph DB means looking for a pattern in the graph (i.e., in the unique model)

A KG can be seen as a representative of a set of KGs, those corresponding to its models. Querying a KG means looking for a pattern in many graphs

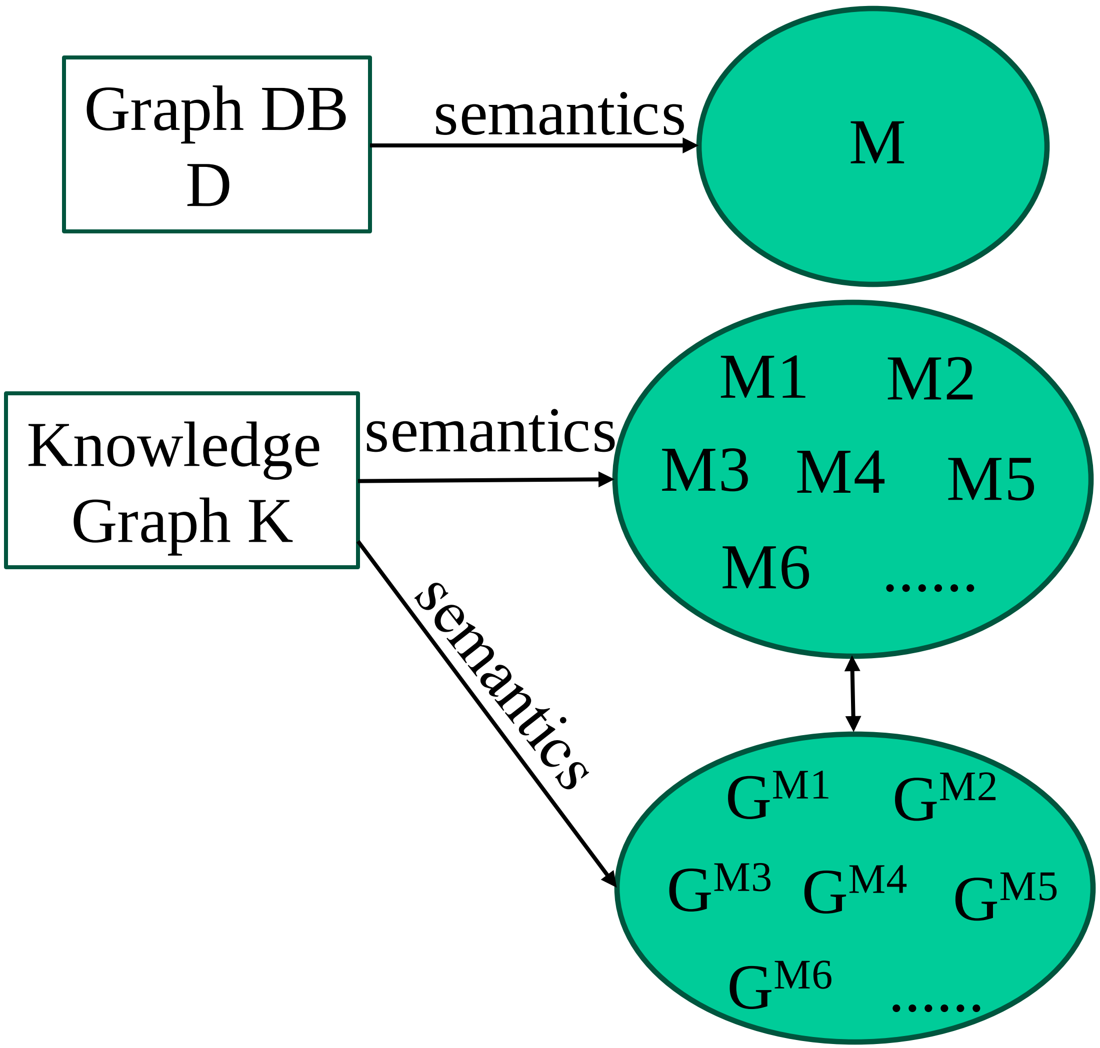
The notion of query evaluation in KGs

A graph database simply corresponds to one model, and we understand well the notion of (and we know how to compute) the answer of a Basic Graph Pattern (BGP) SPARQL query over a graph G (hereby called “DB answer”), obtained in particular by homomorphisms from the query to G (i.e., the unique model of G).

On the other hand, a KG has, in general, many models, and therefore it is not clear how to evaluate queries over KGs. The solution to this problem is to resort to the so-called certain answer semantics. We already noticed that every model of a KG can be seen as a KG itself. This exploited as follows:

Let Q be a BGP SPARQL query and let G be KG. A tuple A of URI is a **certain answer to Q over G** if there exists a function h from Q to G such that h is a homomorphism from Q to every model of G and A is the projection of h to the target variables of Q.

Differently from a DB, a KG has many models



A is a DB answer to Q over D if A is obtained by a homomorphism from Q to D (or, to M, the unique model of D).

A is a certain answer to Q over K if there exists a homorphism h from Q to G^{M1}, and also from Q to G^{M2} , and also from Q to G^{M3}, and also ...... such that A is the projection of h to the target variables of Q

How to compute certain answers

But how to compute the certain answers to a BGP SPARQL query over a KG K? We now introduce the notion of completion of K, denoted *comp*(K) – (here, x,y are URI or blank nodes appearing in K):

comp(K) = *IBC*(K)

$\cup$ { (x **`rdfs:subClassOf`** y) | K $\models$ (x **`rdfs:subClassOf`** y) }

$\cup$ { x **`rdfs:subPropertyOf`** y | K $\models$ x **`rdfs:subPropertyOf`** y) }

$\cup$ { (x **`rdfs:domain`** y) | K $\models$ (x **`rdfs:domain`** y) }

$\cup$ { (x **`rdfs:range`** y) | K $\models$ (x **`rdfs:range`** y) }

The next theorem provides the solution to this problem.

Theorem If Q is a BGP SPARQL query and K is a KG, then the set of certain answers to Q over K is the set of DB answers to Q over *comp*(K), seen as a graph database.

The proof of the theorem is left as a (difficult) exercise.

How to compute certain answers

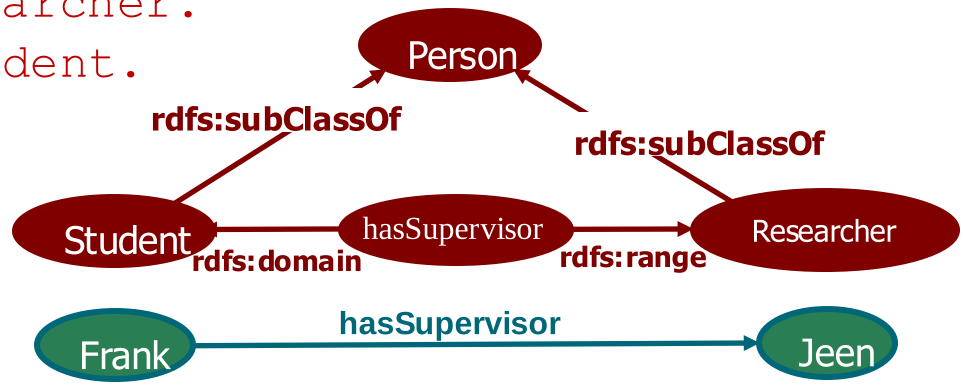
We repeat the theorem of the previous slides:

Theorem If Q is a BGP SPARQL query and K is a KG, then the set of certain answers to Q over K is the set of DB answers to Q over *comp*(K), seen as a graph database.

The above theorem is extremely important: it allow us to conclude that in order to compute the certain answers to Q over K, we can simply compute *comp*(K), and then evaluate Q over *comp*(K) as if it were a simple graph DB, i.e., compute the DB answers to Q over *comp*(K) .

Querying KGs: example

```
KG K:
Student rdfs:subClassOf Person.
Researcher rdfs:subClassOf Person.
hasSupervisor rdfs:range Researcher.
hasSupervisor rdfs:domain Student.
Frank hasSupervisor Jeen.
```




```
Query Q:

select ?x
from G
where { (?x ?y ?z). (?x rdf:type Person). (?z rdf:type ?w).
        (?y rdfs:range ?w).
```

Certain answers: ?

Querying KGs: example

KG K:

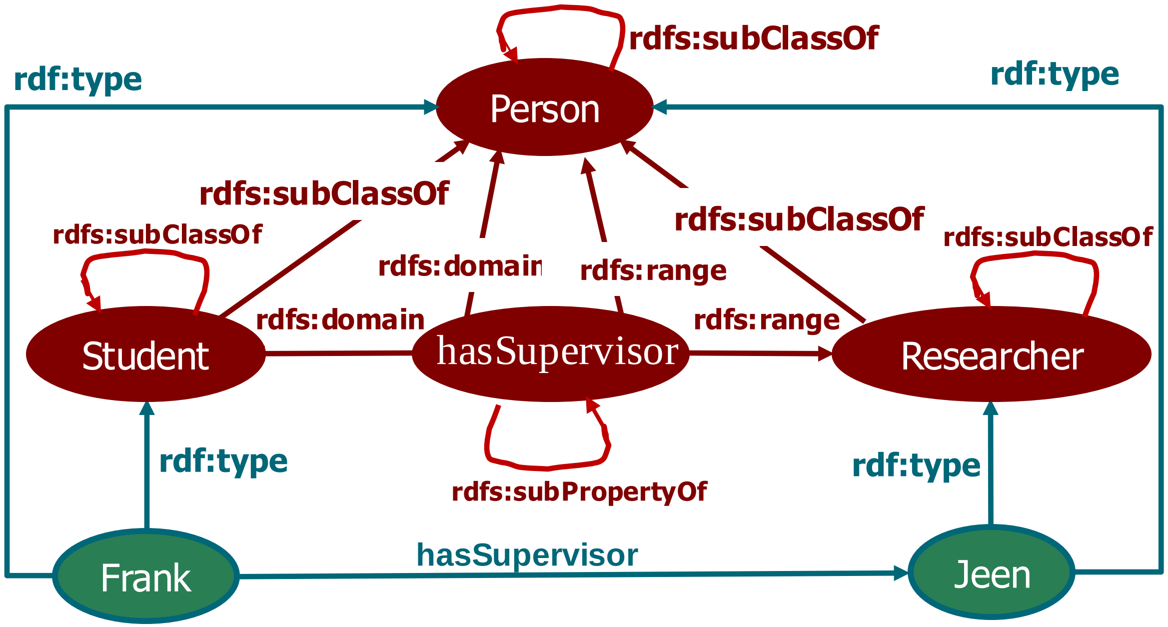
Student **rdfs:subClassOf** Person.

Researcher **rdfs:subClassOf** Person.

hasSupervisor **rdfs:range** Researcher.

hasSupervisor **rdfs:domain** Student.

Frank **hasSupervisor** Jeen.



Comp(K)

```
Query Q:

select ?x
from G
wwhere { (?x ?y ?z). (?x rdf:type Person). (?z rdf:type ?w).
        (?y rdfs:range ?w).
```

Certain answers: { Frank }