

# Data Management for Data Science

*Corso di laurea magistrale in Data Science  
Sapienza Università di Roma  
2017/2018*

## Graph Databases

**Domenico Lembo, Riccardo Rosati**

*Dipartimento di Ingegneria Informatica Automatica e Gestionale A. Ruberti  
Sapienza Università di Roma*

# Credits

---

- Some examples and figures on graph databases are taken from: Ian Robinson, Jim Webber, & Emil Eifrem. Graph Databases. O'Reilly. 2013. Available at <http://graphdatabases.com/>
- The slides from 18 to 30 are taken from: Maribel Acosta, Cosmin Basca, Alejandro Flores,, Edna Ruckhaus, Maria-Esther Vidal. Semantic Data Management in Graph Databases. ESWC-14 tutorial ([ABFRV14]), with minor adaptations.
- The part on RDF storage is taken from: Yongming Luo, Francois Picalausa, George H.L. Fletcher, Jan Hidders, and Stijn Vansummeren. Storing and Indexing Massive RDF Data Sets. In Semantic Search over the Web. Springer. 2012

# Graph Databases

- **Introduction to Graph Databases**
- Resource Description Framework (RDF)
- Querying RDF databases: The SPARQL language
- RDF storage
- Linked data
- Tools

# The NoSQL movement

---

- Since the 80s, the dominant **back end** of business systems has been a **relational database**
- It's remarkable that many architectural variations have been explored in the design of clients, front ends, and middle-ware, on a multitude of platforms and frameworks, but haven't until recently questioned the architecture of the back end.
- In the past decade, we've been faced with **data that is bigger in volume, changes more rapidly, and is more structurally varied** (in a definition, Big Data) than can be dealt with by traditional RDBMS deployments.
- The NOSQL 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** => Big data may be regularly or irregularly structured, dense or sparse, connected or disconnected
- **Query execution times increase as the size of tables** and the number of joins grow (so-called *join pain*) => this is not sustainable when we require sub-second response to queries (NoSQL approaches tend to organize the data in such a way that the join is already computed, but this comes at the price of flexibility: essentially no precomputed joins cannot be executed)

# Graph databases

---

- A graph database is a database that uses **graph structures** 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.
- Graph databases can be used for both **OLAP** (since are naturally multidimensional structures ) and **OLTP**.
- Systems tailored to OLTP (e.g., Neo4j) are generally optimized for *transactional performance*, and tend to guarantee ACID properties.

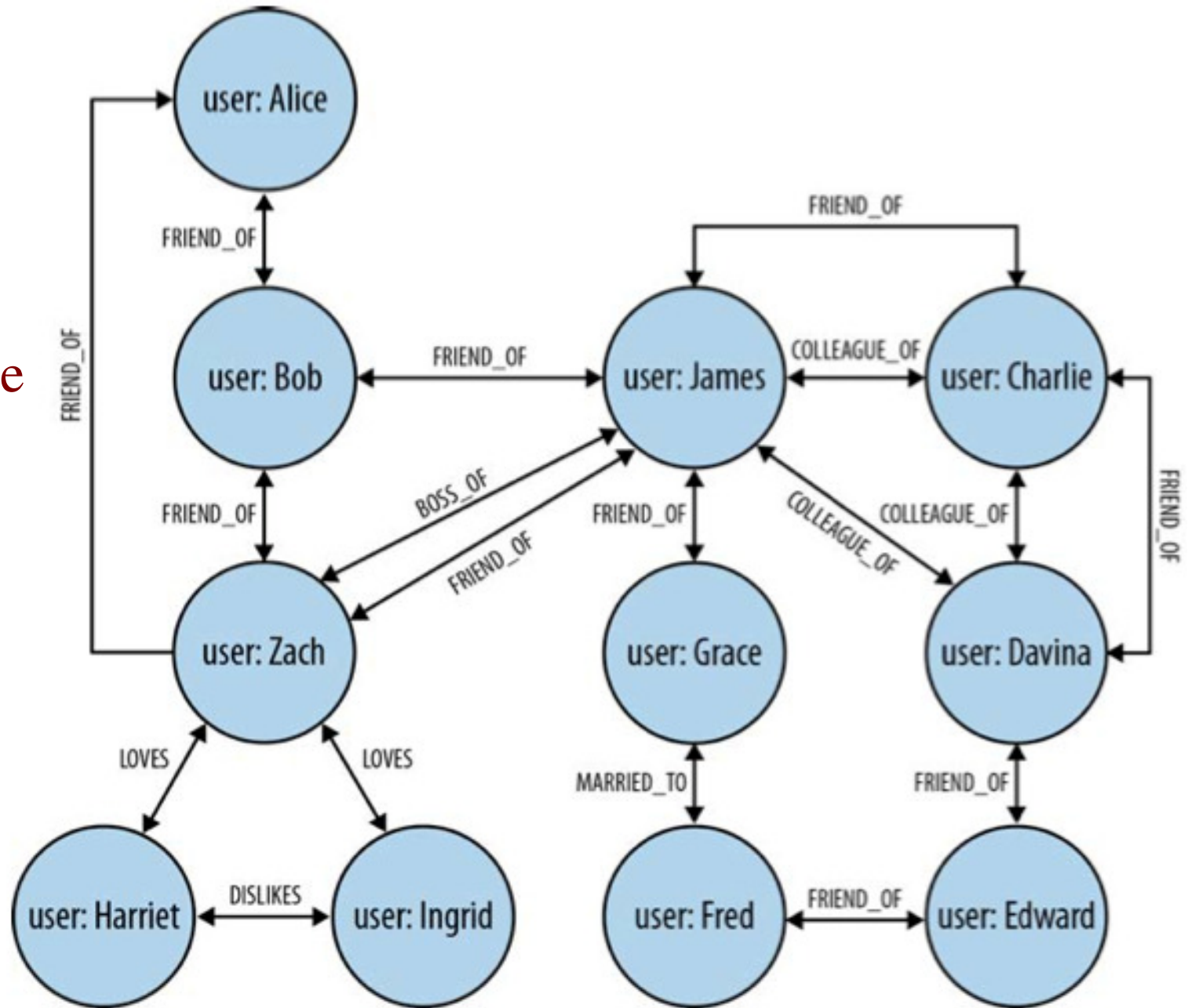
# Graph databases

---

- Graph databases are **schemaless**:
  - Thus they will behave in response to the dynamics of big data: you can accumulate data incrementally, without the need of a predefined, rigid schema;
  - They provide flexibility in assigning different pieces of information with different properties, at any granularity;
  - They are very good in managing sparse data.
  - This does not mean that intensional aspects cannot be casted into a graph, but they are not pre-defined and are normally treated as data are treated.
- Graph databases can be queried through (standardized) languages: depending on the storage engine (see later) 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 dynamic information is natural and simple





# Graph Databases Embrace Relationships

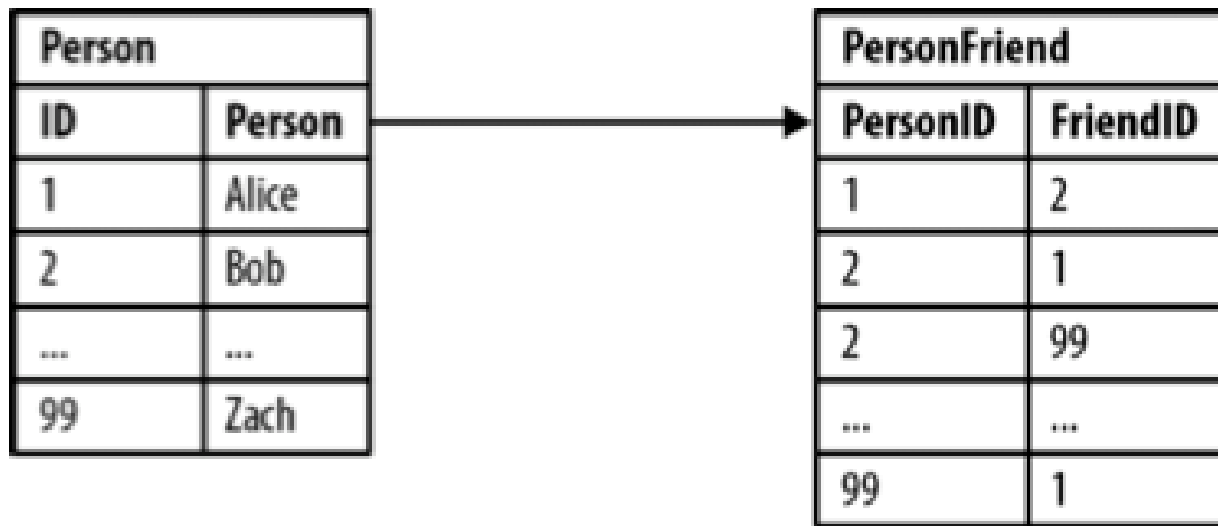
---

- Obviously, graph databases are particularly suited to model situations in which the information is somehow “natively” in the form of a graph.
- The real world provide us with a lot of application domains: social networks, recommendation systems, geospatial applications, computer networks and data center management, authorization and access control systems, to mention some of them.
- The success key of graph databases in these contexts is the fact that they provide **native means to represent relationships**.
- Relational databases instead lack relationships: they have to be simulated through the help of foreign keys, thus adding additional development and maintenance overhead, and “discover” them require costly join operations.

# Graph DBs vs Relational DBs- Example

---

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

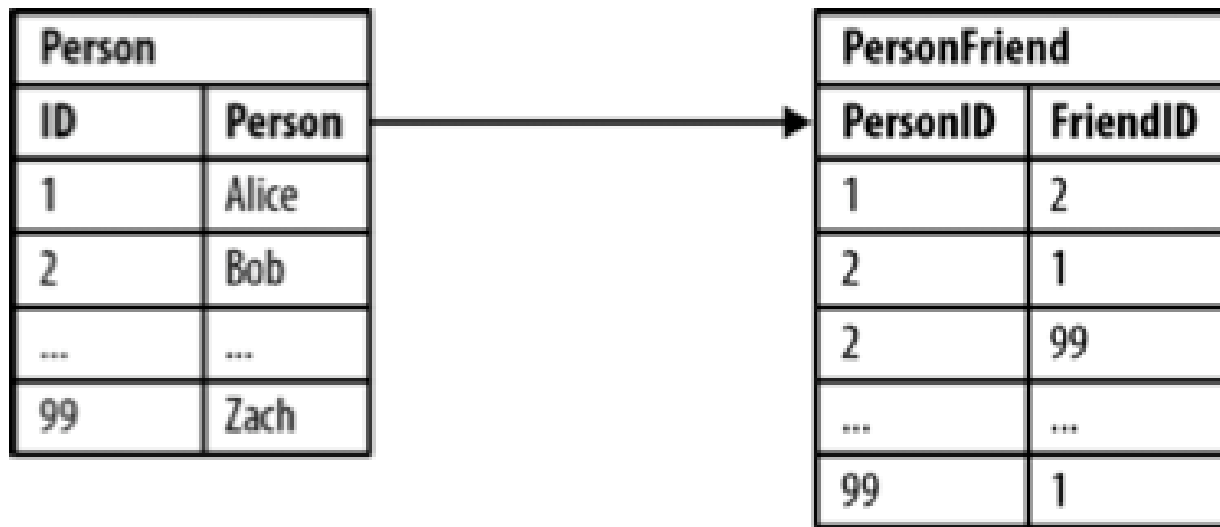


Notice that PersonFriend has not to be considered as symmetric: Bob may consider Zach as friend, but the converse does not necessarily hold

# Graph DBs vs Relational DBs- Example

---

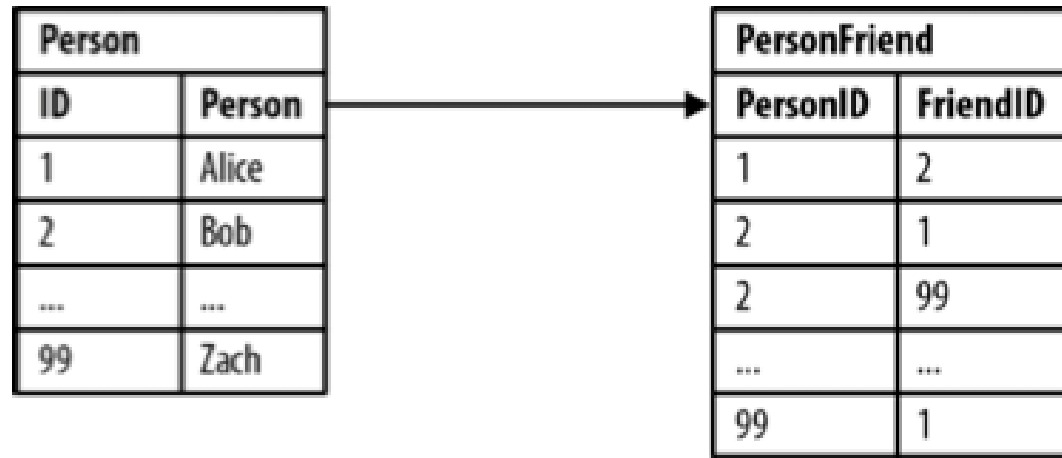
Asking “who are Bob’s friends?” (i.e., those that Bob considers as friend) is easy



```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend ON
    PersonFriend.FriendID = p1.ID JOIN Person p2 ON
    PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

# Graph DBs vs Relational DBs- Example

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



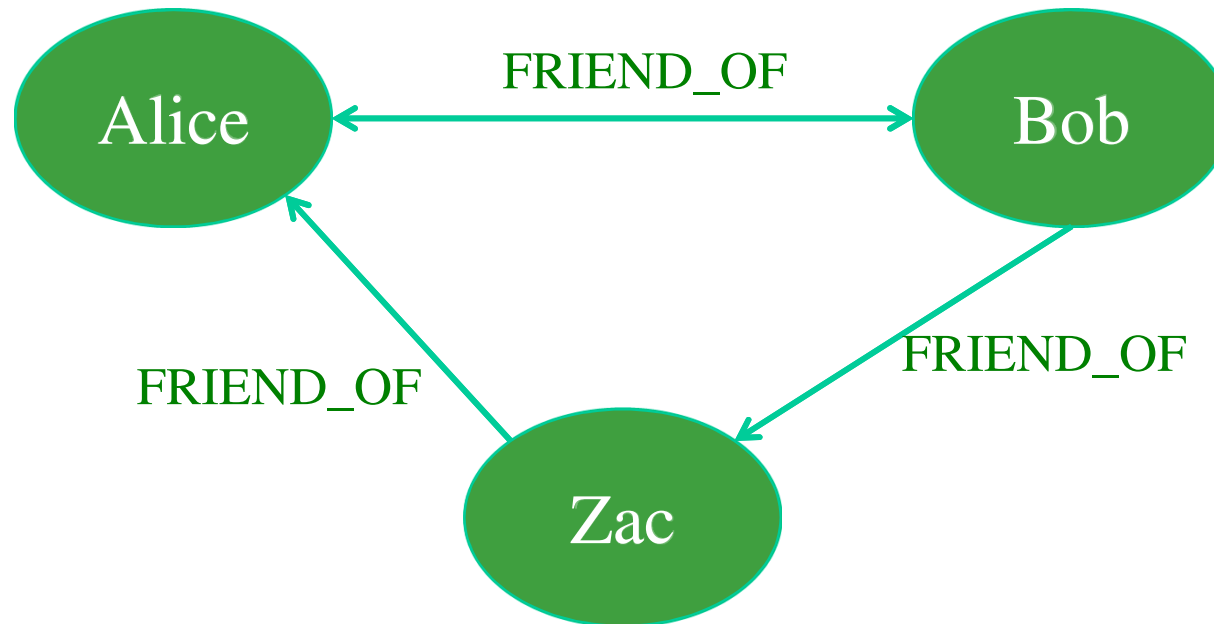
```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1 ON
    pf1.PersonID = p1.ID JOIN PersonFriend pf2 ON
    pf2.PersonID = pf1.FriendID JOIN Person p2 ON
    pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

*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 (but other operations may be however more difficult)

# Graph DBs vs Relational DBs- Experiment

---

The following table reports result of an experiment aimed to finding 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 chosen at random, 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- Experiment

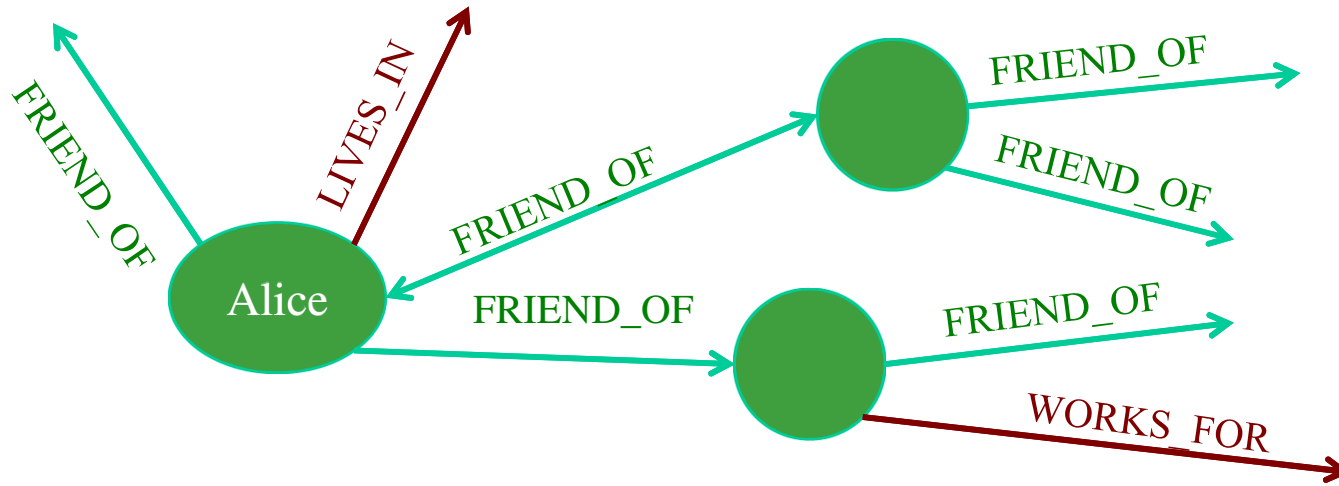
---

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

- Indeed, in the SQL query we need *3 joins* (each table is joined twice).
- If  $n$  is the number of persons and  $m$  is the number of pairs of friends, this means a cost of  $O(n^2m^2)$ .
- **Indexes reduce this cost**, since they allow us to avoid linear search over a column.
- Assuming that the structure of the index is a binary search tree, the cost is  $O((\log_2 n)^2 (\log_2 m)^2)$

# Graph DBs vs Relational DBs- Experiment

---



- Starting from a node, we have to scan all outgoing edges to identify FRIEND\_OF edges
- We then have to traverse the edges and repeat the search in all the reached nodes.
- If  $x$  bounds the number of outgoing edges (assuming  $x \ll n$ ), and  $k$  bounds the number of FRIEND\_OF edges outgoing from a node ( $k \ll m$ ), then the cost here is  $O(x) + O(kx)$  (the cost of traversal is constant)
- Local indexes are normally used to speed up local search



# Graph DBs vs Relational DBs- Queries\*

---

- **Relational Databases** (querying is through joins)
  - In effect, **the join operation forms a graph that is dynamically constructed** as one table is linked to another table. While having the benefit of being able to dynamically construct graphs, 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 friend's), **all data in all queried tables must be examined** 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 Multi-Graph

$G=(V,E,\Sigma,L)$  is a multi-graph:

- $V$  is a finite set of nodes or vertices,  
e.g.  $V=\{\text{Term, forOffice, Organization, ...}\}$
- $E \subseteq V \times V$  is a set of edges representing **binary** relationship between elements in  $V$ ,

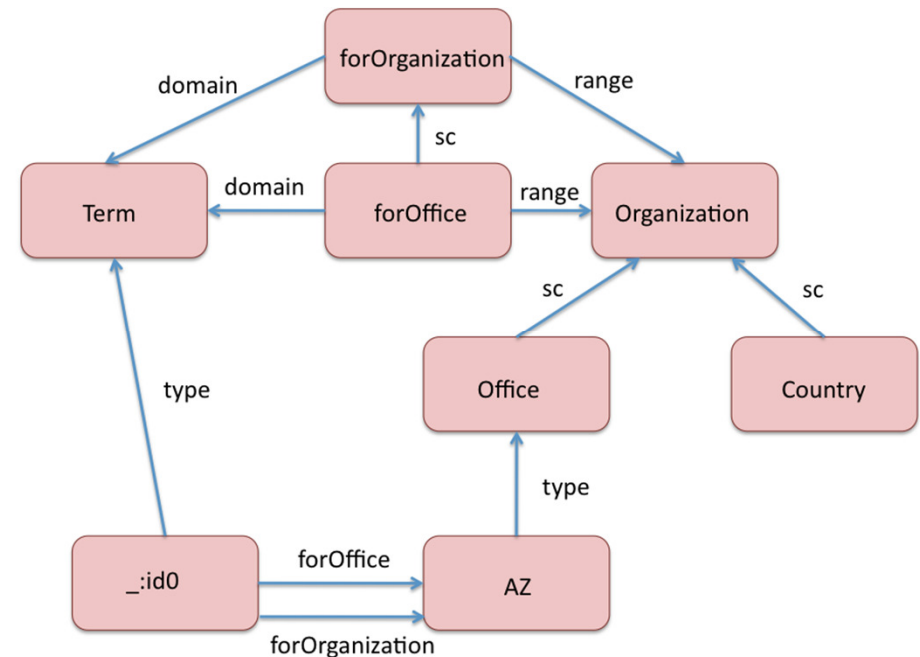
e.g.  $E=\{(\text{forOffice,Term})$   
 $(\text{forOffice,Organization}),(\text{Office,Organization})$   
 $\dots\}$

- $\Sigma$  is a set of labels,

e.g.,  $\Sigma=\{\text{domain, range, sc, type, ...}\}$

- $L$  is a function:  $V \times V \rightarrow \text{PowerSet}(\Sigma)$ ,

e.g.,  $L=\{((\text{forOffice,Term}),\{\text{domain}\}), ((\text{forOffice,Organization}),\{\text{range}\}),$   
 $((\_id0,AZ),\{\text{forOffice, forOrganization}\})\dots\}$



# Basic Operations

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

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

# Implementation of Graphs

[Sakr and Pardede 2012]

## 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.

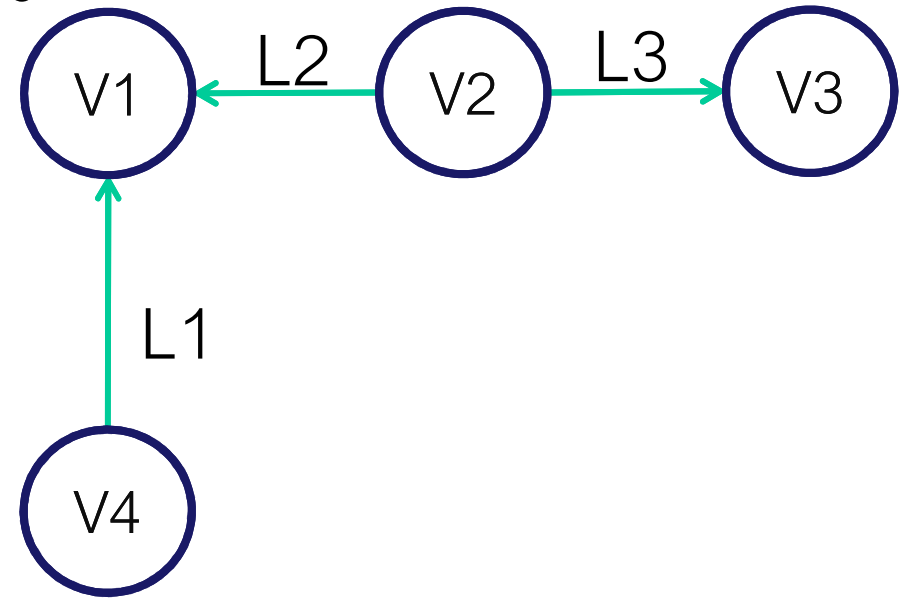
From [ABFRV14]

# Adjacency List

V1	
V2	
V3	
V4	

→ (V1, {L2}) (V3, {L3})

→ (V1, {L1})

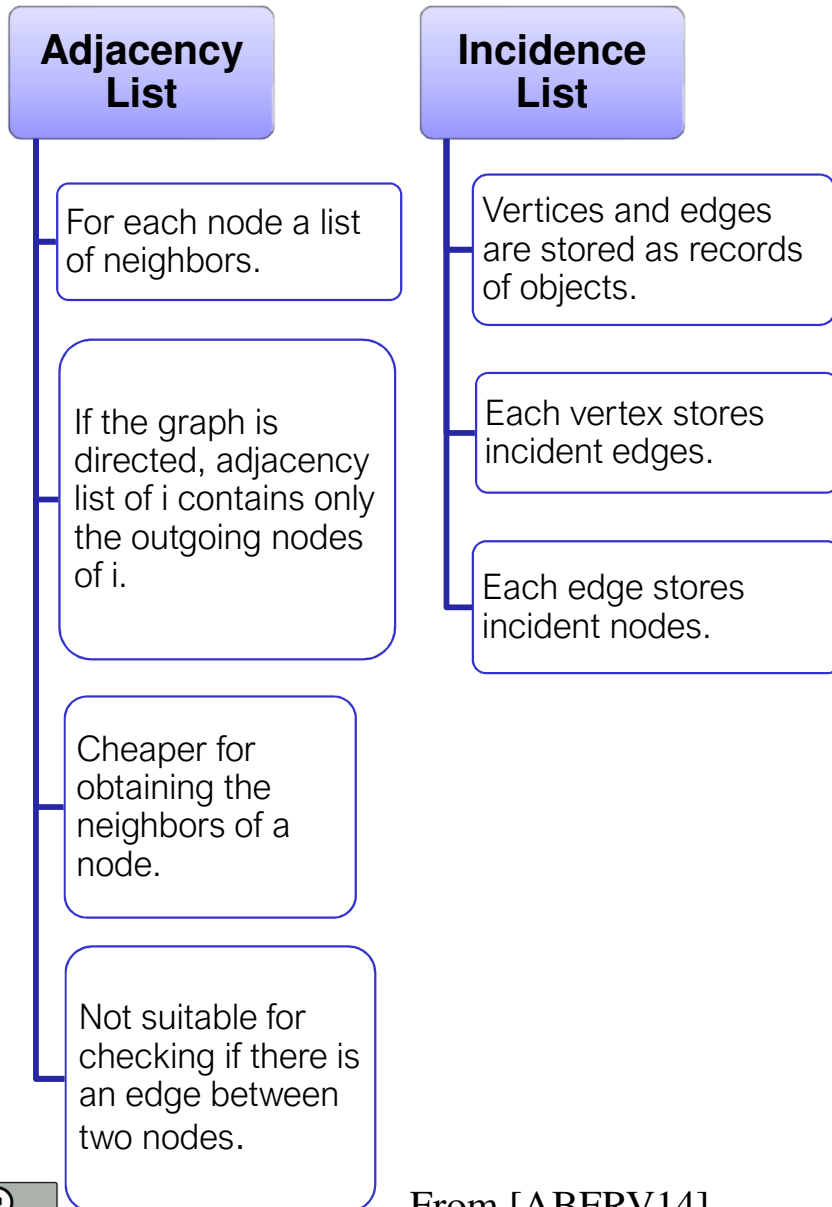


## Properties:

- Storage:  $O(|V|+|E|+|L|)$
- $\text{Adjacent}(G,x,y)$ :  $O(|V|)$
- $\text{Neighbors}(G,x)$ :  $O(|V|)$
- $\text{AdjacentEdges}(G,x,y)$ :  $O(|V|+|E|)$
- $\text{Add}(G,x,y,l)$ :  $O(|V|+|E|)$
- $\text{Delete}(G,x,y,l)$ :  $O(|V|+|E|)$

# Implementation of Graphs

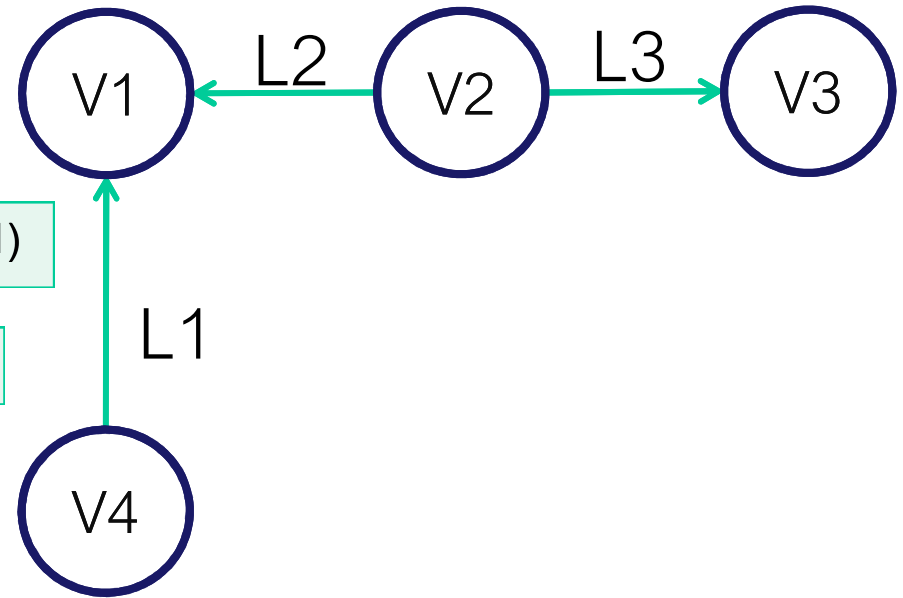
[Sakr and Pardede 2012]



From [ABFRV14]

# Incidence List\*

\* Simplified version: each edge has a different label



V1		→ (destination,L2)	(destination,L1)
V2		→ (source,L2)	(source,L3)
V3		→ (destination,L3)	
V4		→ (source,L1)	

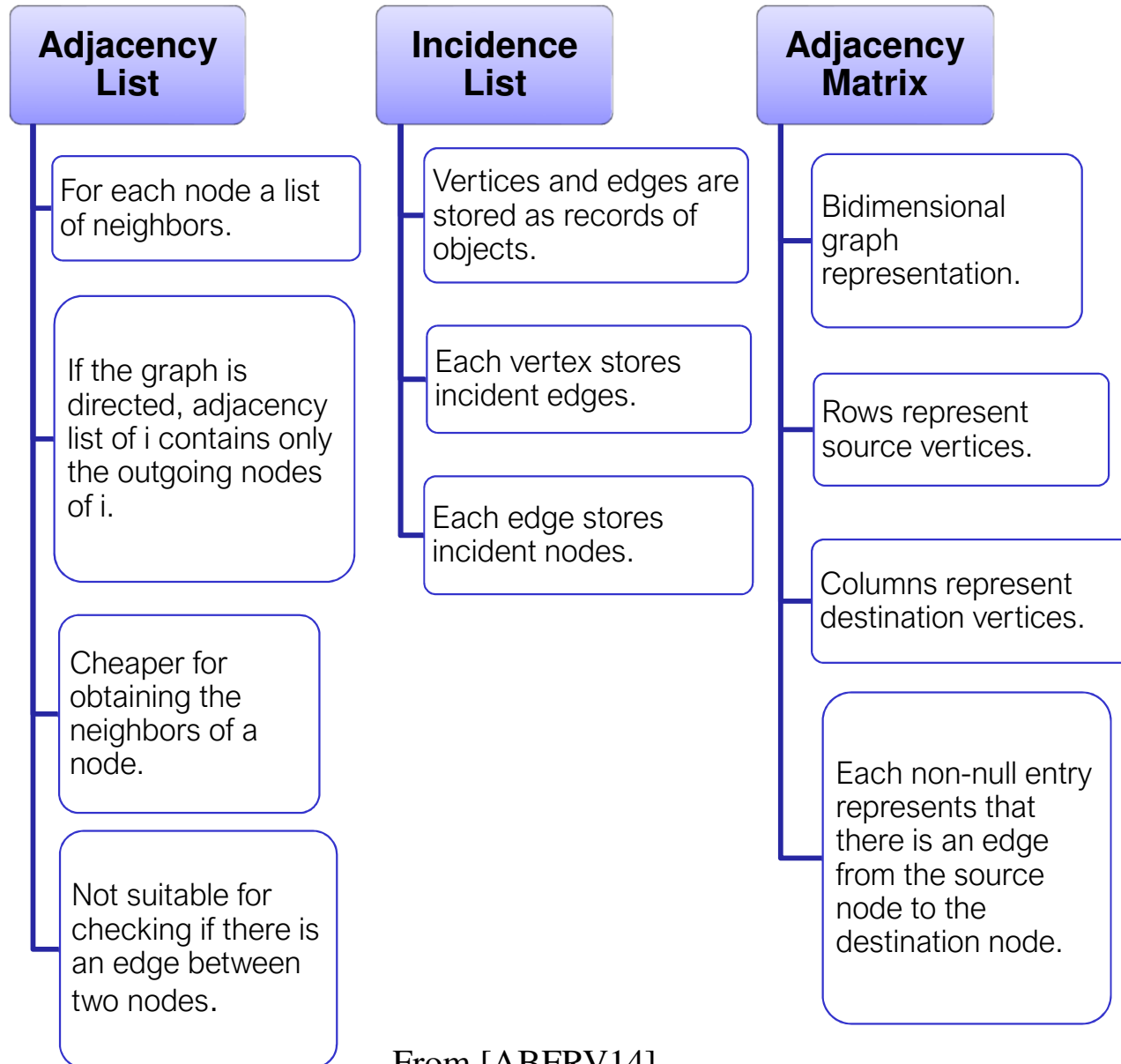
L1		→ (V4,V1)
L2		→ (V2,V1)
L3		→ (V2,V3)

## Properties:

- Storage:  $O(|V|+|E|+|L|)$
- $\text{Adjacent}(G,x,y): O(|E|)$
- $\text{Neighbors}(G,x): O(|E|)$
- $\text{AdjacentEdges}(G,x,y): O(|E|)$
- $\text{Add}(G,x,y,l): O(|E|)$
- $\text{Delete}(G,x,y,l): O(|E|)$

# Implementation of Graphs

[Sakr and Pardede 2012]

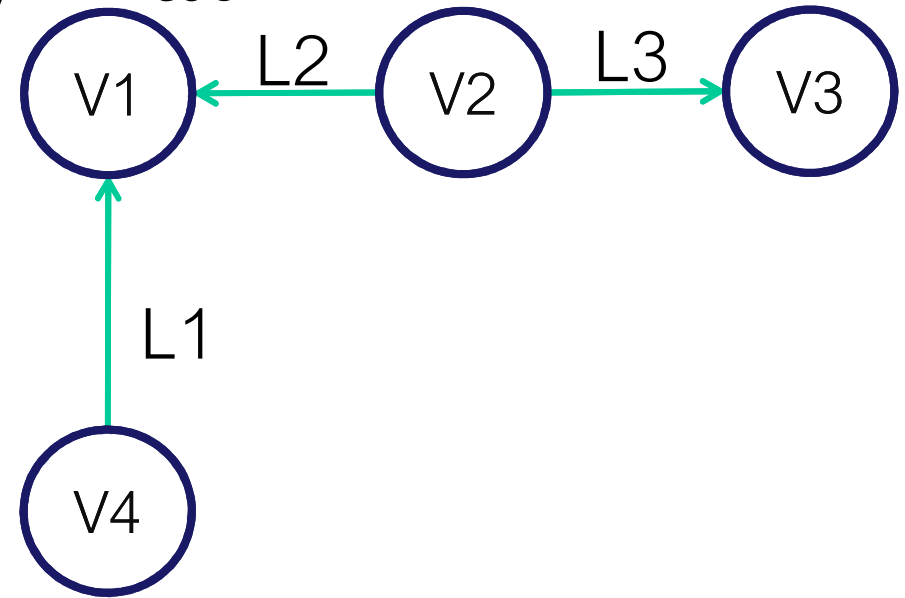


From [ABFRV14]



# Adjacency Matrix

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

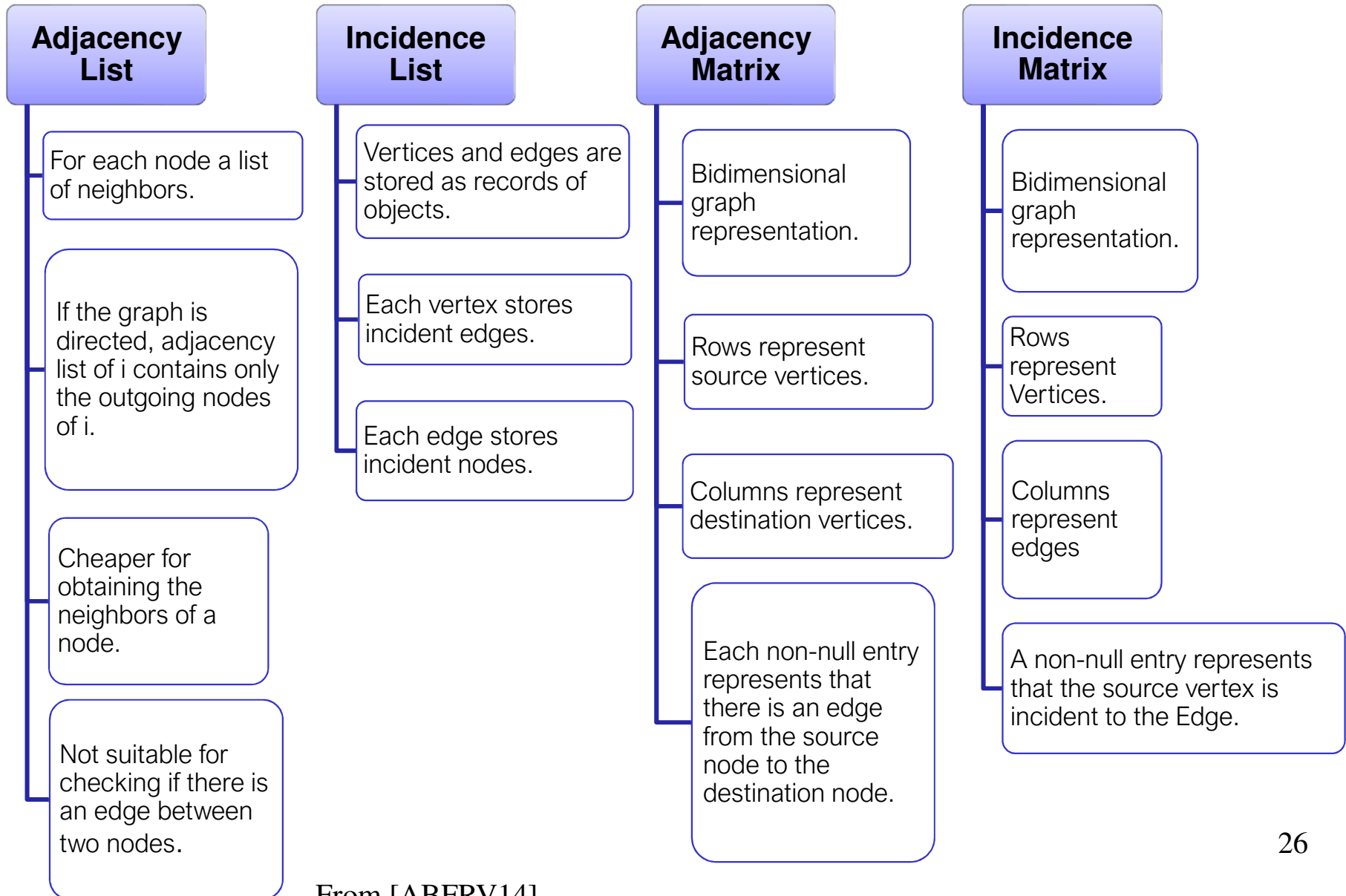


## Properties:

- Storage:  $O(|V|^2)$
- $\text{Adjacent}(G,x,y)$ :  $O(1)$
- $\text{Neighbors}(G,x)$ :  $O(|V|)$
- $\text{AdjacentEdges}(G,x,y)$ :  $O(|E|)$
- $\text{Add}(G,x,y,l)$ :  $O(|E|)$
- $\text{Delete}(G,x,y,l)$ :  $O(|E|)$

# Implementation of Graphs

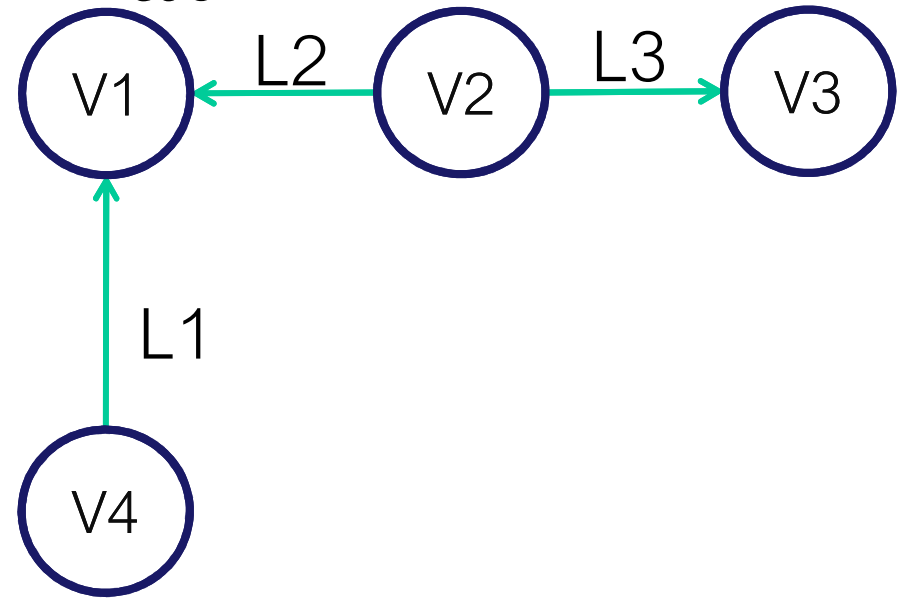
[Sakr and Pardede 2012]



From [ABFRV14]

# Incidence Matrix

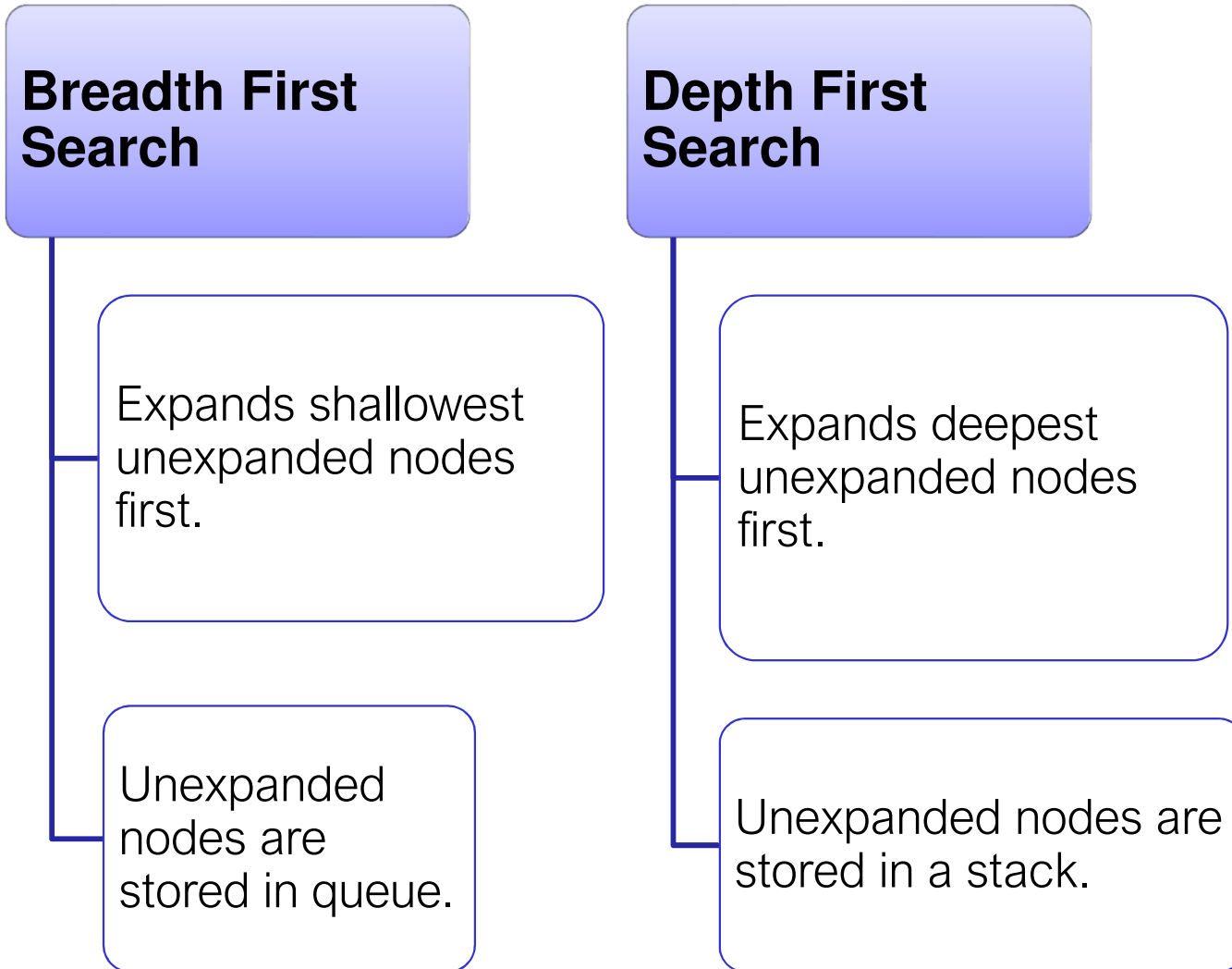
	L1	L2	L3
V1	destination	destination	
V2		source	source
V3			destination
V4	source		



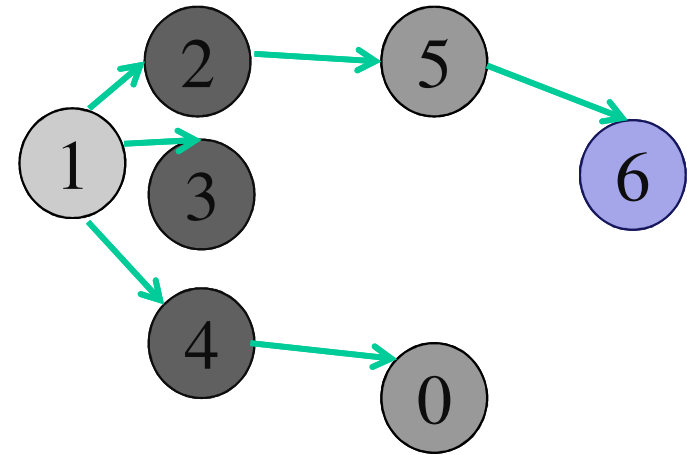
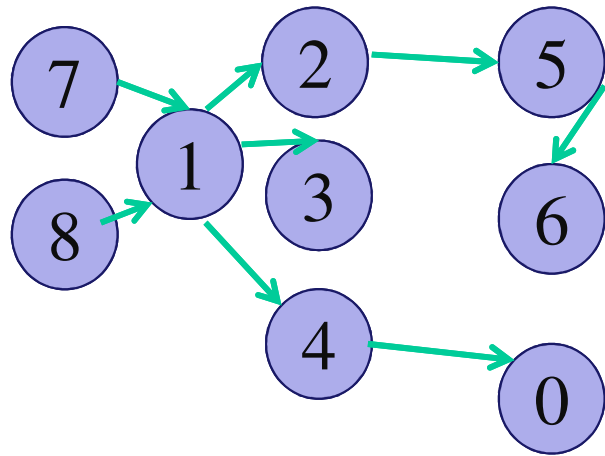
## Properties:

- Storage:  $O(|V| \times |E|)$
- $\text{Adjacent}(G, x, y)$ :  $O(|E|)$
- $\text{Neighbors}(G, x)$ :  $O(|V| \times |E|)$
- $\text{AdjacentEdges}(G, x, y)$ :  $O(|E|)$
- $\text{Add}(G, x, y, l)$ :  $O(|V|)$
- $\text{Delete}(G, x, y, l)$ :  $O(|V|)$



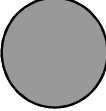
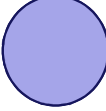
# Traversal Search



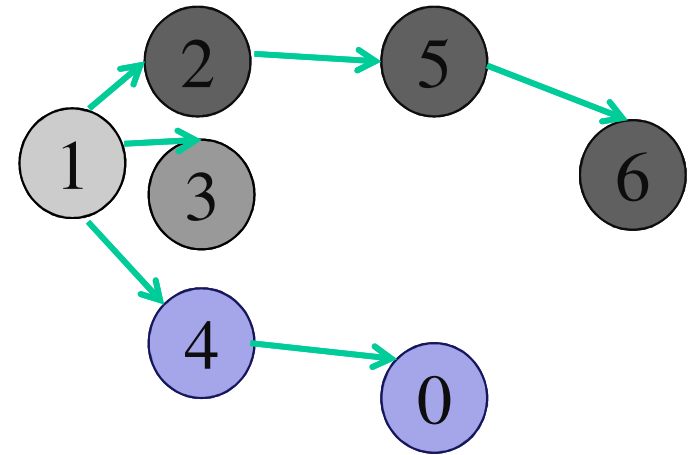
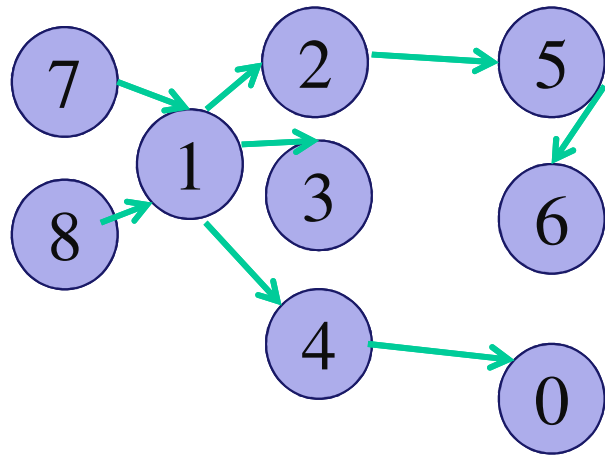
# Breadth First Search



## Notation:

-  Starting Node
-  First Level Visited Nodes
-  Second Level Visited Nodes
-  Third Level Visited Nodes

# Depth First Search



## Notation:

- ① Starting Node
- First Level Visited Nodes
- Second Level Visited Nodes
- Third Level Visited Nodes

# 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  $\Sigma$  consists of a finite set  $V$  of nodes and a set of **labeled edges**  $E \subseteq V \times \Sigma \times V$ .
- a path  $\pi$  in  $G$  from node  $v_0$  to node  $v_m$  is a sequence of the form  $(v_0, a_1, v_1)(v_1, a_2, v_2) \dots (v_{m-1}, a_m, v_m)$ , where  $(v_{i-1}, a_i, v_i)$  is an edge in  $E$ , for each  $1 \leq i \leq m$ . The *label* of  $\pi$ , denoted  $\lambda(\pi)$ , is the string  $a_1 a_2 \dots a_m \in \Sigma^*$ .
- A *Regular path query* is a **regular expression**  $L$  over  $\Sigma$ . 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  $\pi$  in  $G$  from  $u$  to  $v$  such that  $\lambda(\pi)$  satisfies  $L$ .
- **Query languages for graph databases normally extend this class of queries**

# Graph Database Management Systems\*

---

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



\*From Graph Database Management Systems. Course on Big Data, prof. Riccardo Torlone (Univ. Roma Tre), available at [www.dia.uniroma3.it/~torlone/bigdata/materiale.html](http://www.dia.uniroma3.it/~torlone/bigdata/materiale.html)



# 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.
- 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 technologies 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 GDBMS 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

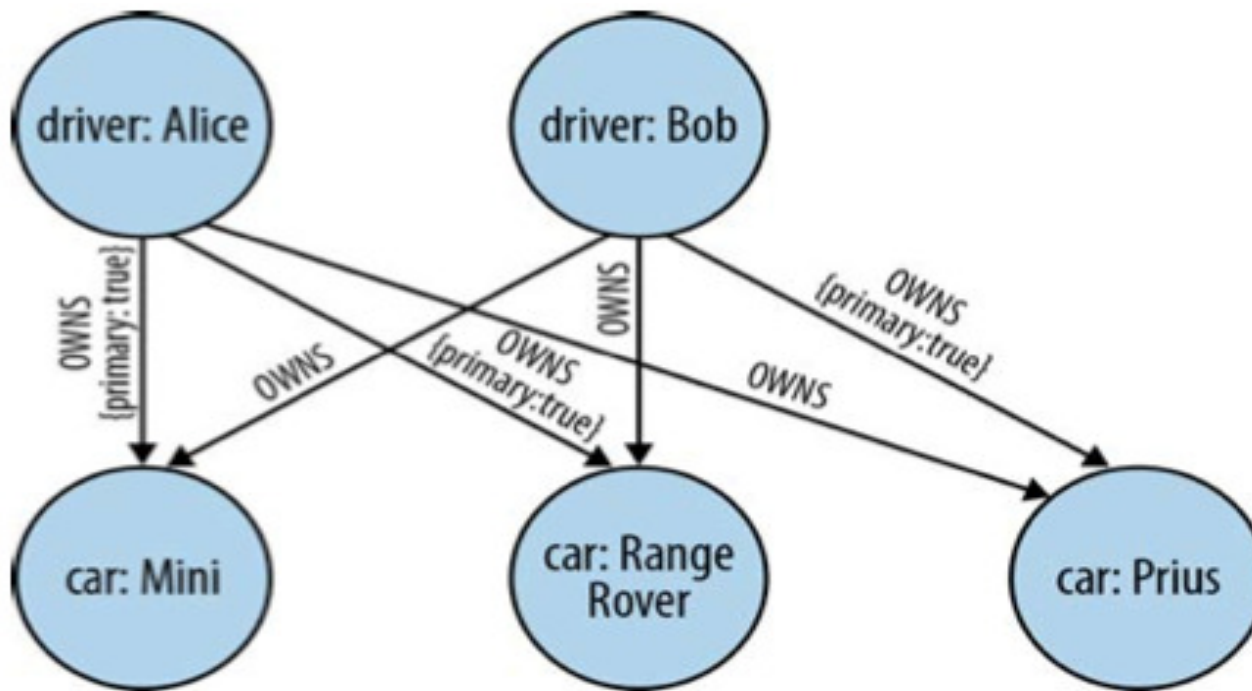
---

- There are several different graph data models, which somehow generalizes the basic definition we have seen before, including
  - property graphs,
  - hypergraphs,
  - triple stores.

# Property-graph databases

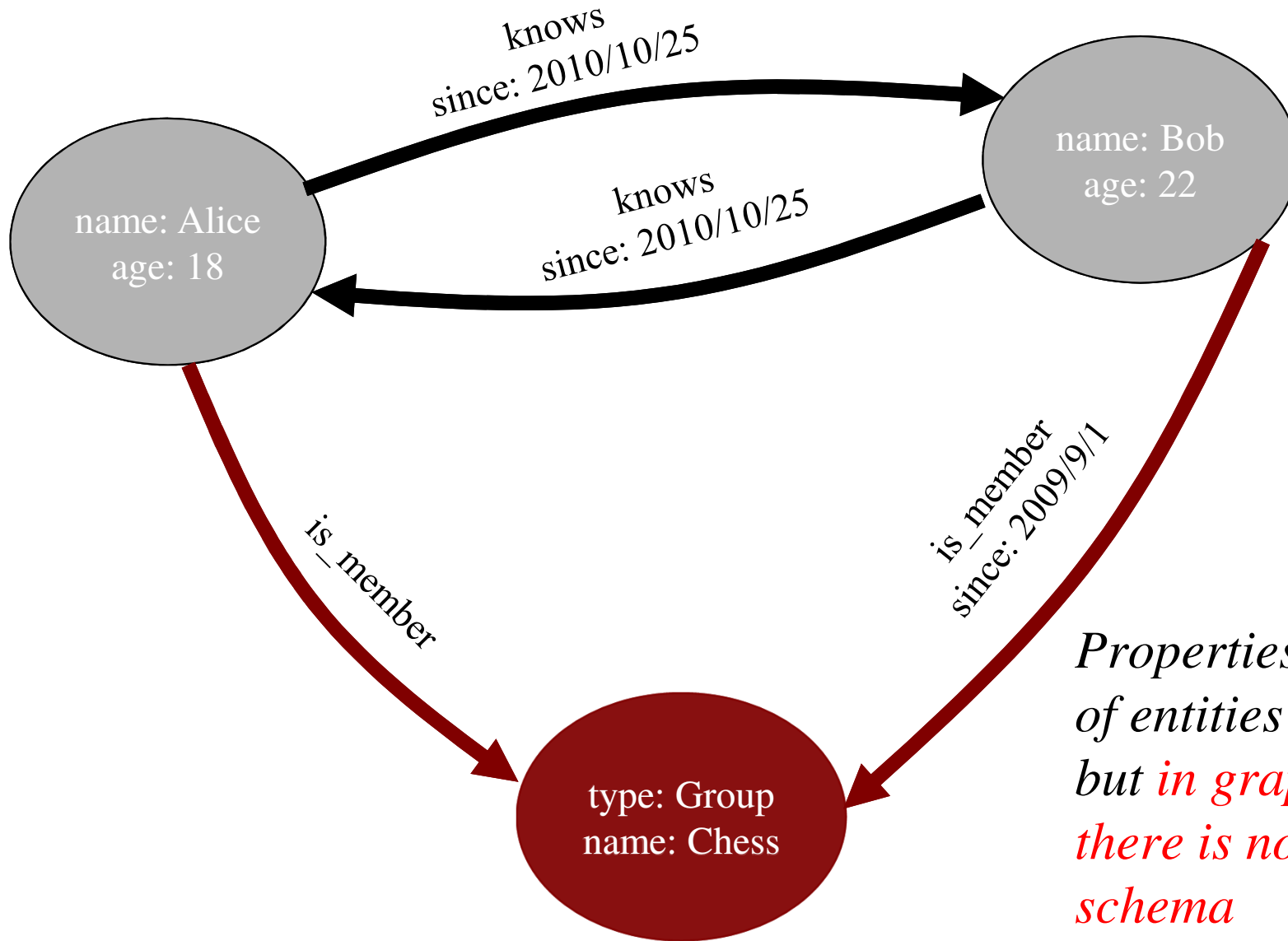
---

- A property graph is a labeled directed multigraph  $G = (V, E)$  where every node  $v \in N$  and every edge  $e \in E$  can be associated with a set of *<key, value>* pairs, called *properties*.
- Each edge represents a relationship between nodes and is associated with a label, which is the name of the relationship.



# Property-graph databases

---



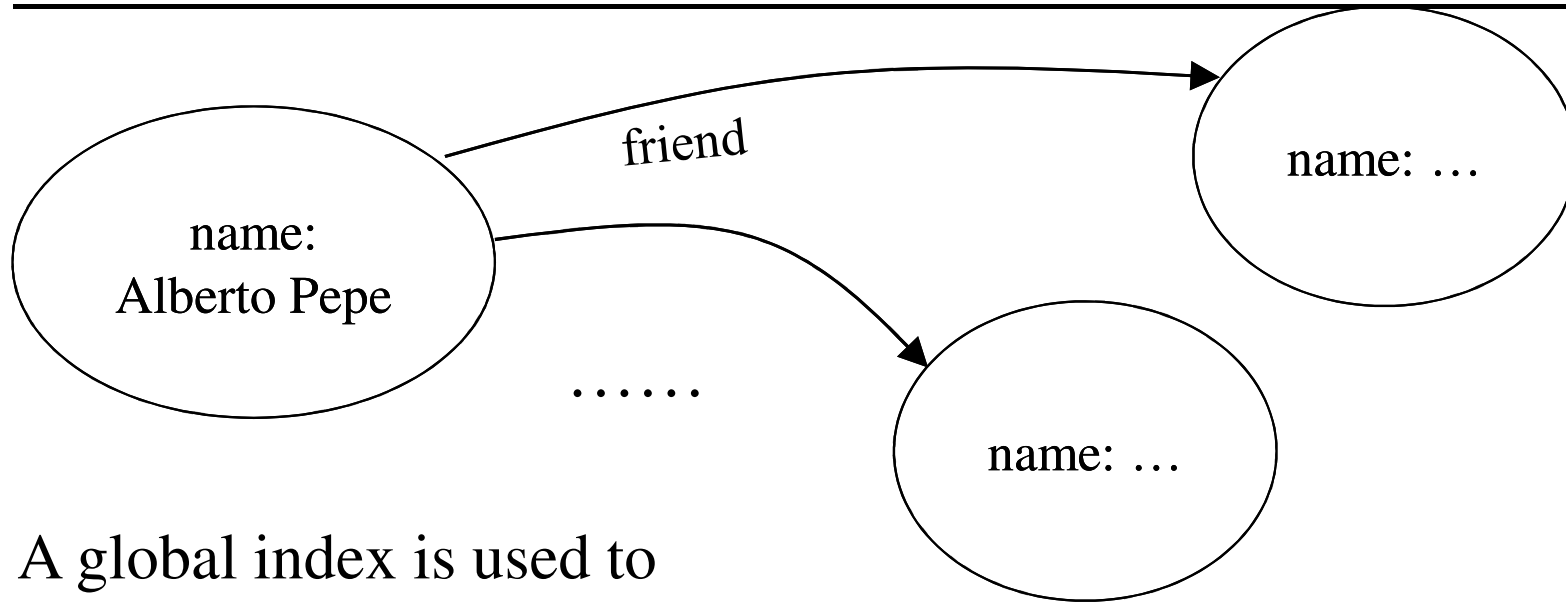
*Properties act as attributes of entities or relationships, but **in graph databases** there is no a-priori or rigid schema*

# Querying property-graph databases

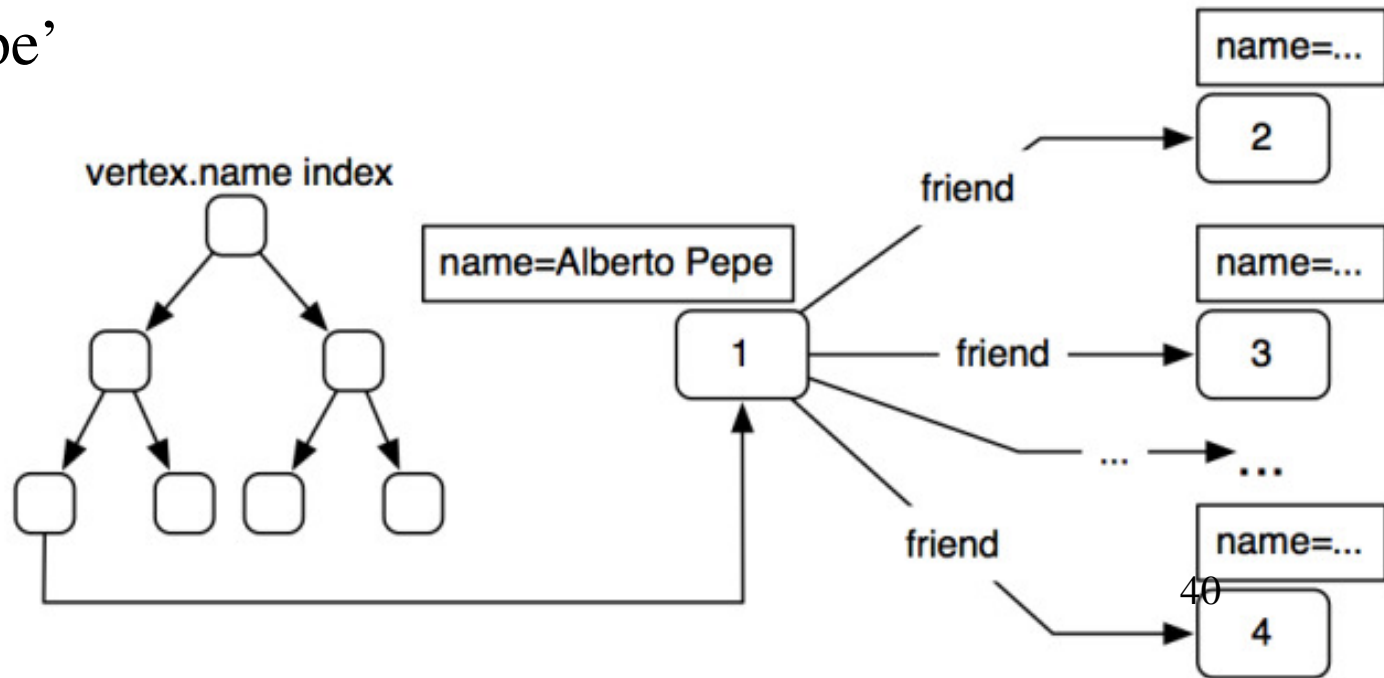
---

- 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 accessing properties, however, besides exploiting adjacency, somehow relies on relational mechanisms:
  - In the property graph model, it is common for the properties of the vertices (and sometimes edges) to be indexed using a tree structure analogous, in many ways, to those used by relational databases.

# People and their friends example

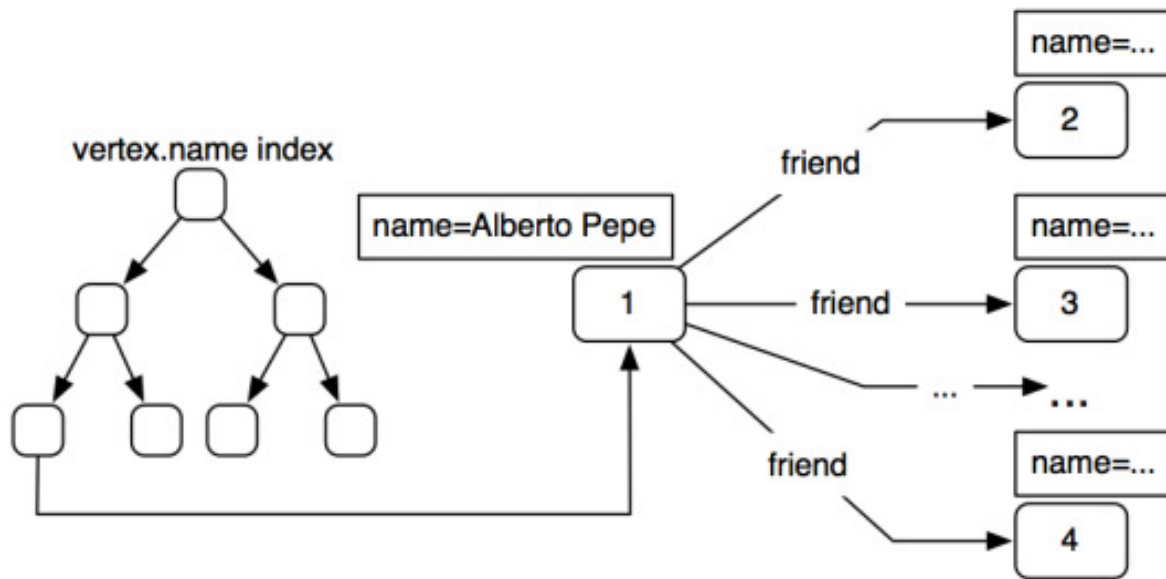


A global index is used to access node whose name property is 'Alberto Pepe'





# People and their friends example

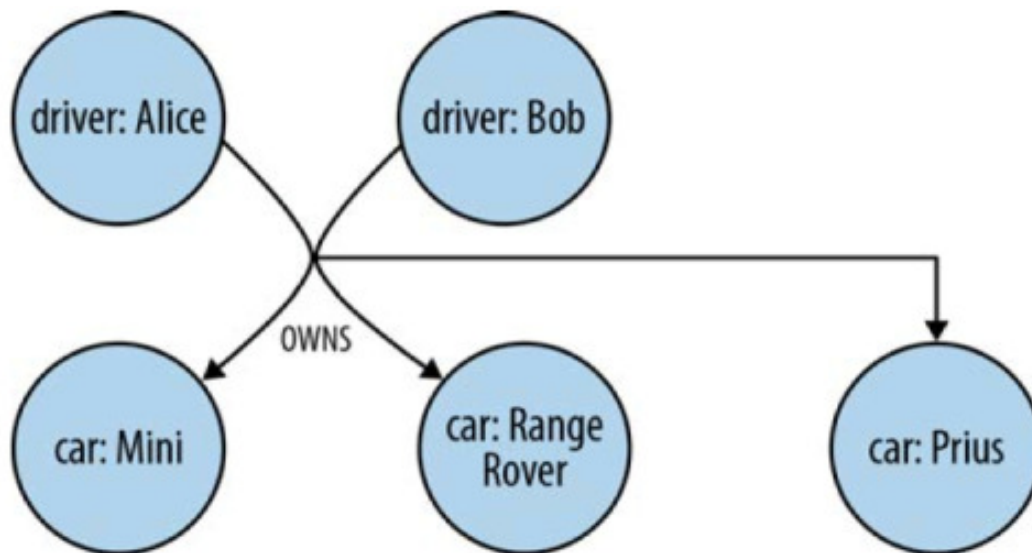


1. Query the vertex.name index to find all the vertices with the name “Alberto Pepe” [ $O(\log_2 n)$ ] (where  $n$  is the number of nodes with the name property)
2. Given the vertex returned, get the  $k$  friend edges emanating from this vertex. [ $O(k + x)$ ] (where  $k$  is the number of friends and  $x$  is the number of the other outgoing edges)
3. Given the  $k$  friend edges retrieved, get the  $k$  vertices on the heads of those edges. [ $O(k)$ ]
4. Given these  $k$  vertices, get the  $k$  name properties of these vertices. [ $O(ky)$ ] (where  $y$  is the number of properties in each vertex)

# 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 many-to-many relationships
- 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 lose some flexibility in specifying some properties** (e.g., who is the primary owner)
- Notice that **any hypergraph database can be encoded into a graph database**



# Triple stores

---

- 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:** structuring information in triples does not per se realize the idea of the Semantic Web, and thus it does not allow for knowledge inference. Nonetheless, triple stores turned out to be a particularly useful format to exchange information on the Web and have become nowadays very popular, not only in the semantic web context.

# Graph Databases

- Introduction to Graph Databases
- **Resource Description Framework (RDF)**
- Querying RDF databases: The SPARQL language
- RDF storage
- Linked data
- Tools

# Resource Description Framework

---

- RDF is a data model
  - ✓ the model is domain-neutral, application-neutral and ready for internationalization
  - ✓ besides viewing it as a graph data model, it can be also viewed as an object-oriented model (object/attribute/value)
- A standard XML syntax exists, which allows to specify RDF databases
  - ✓ RDF data model is an abstract, conceptual layer independent of XML
  - ✓ Consequently, XML is a transfer syntax for RDF, not a component of RDF
  - ✓ In principle, RDF data might also occur in a form different from XML

# XML

---

- XML: eXtensible Mark-up Language
- XML documents are written through a user-defined set of tags
- tags can be used to express additional properties of the various pieces of information (i.e., enriching it with its “meaning”)

# XML: example

---

```
<course date="2014">  
  <title>Big Data Management</title>  
  <teacher>  
    <name>Domenico Lembo</name>  
    <email>lembo@dis.uniroma1.it</email>  
  </teacher>  
  <prereq>none</prereq>  
</course>
```

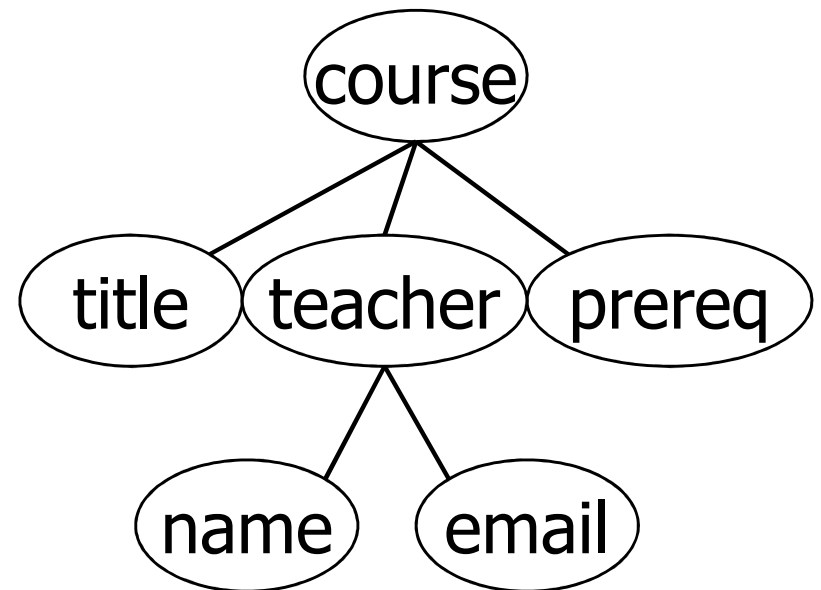
# XML

---

- XML: document = labeled tree
- node = label + attributes/values + contents

```
<course date="...">  
  <title>...</title>  
  <teacher>  
    <name>...</name>  
    <email>...</email>  
  </teacher>  
<prereq>...</prereq>  
</course>
```

=





# RDF model

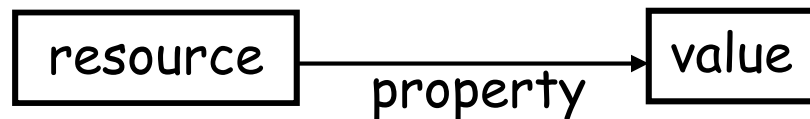
---

RDF model = set of RDF triples

triple = expression (statement)

(subject, predicate, object)

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



# RDF triples

---

example: “the document at

<http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>

has Ora Lassila as creator”

triple:

<http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/> creator “OraLassila”

<http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>

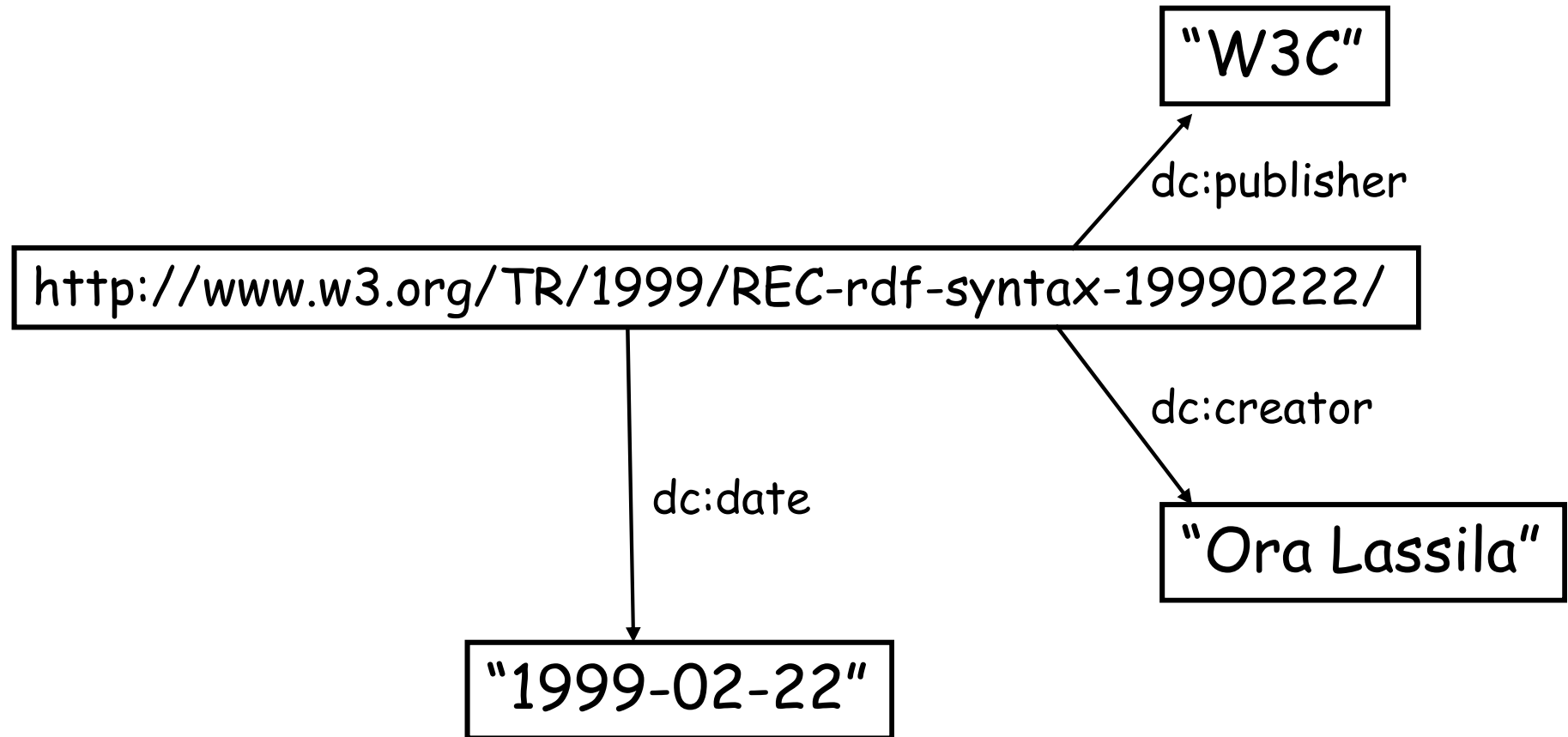
creator

“Ora Lassila”

⇒ RDF model = **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
  - remark: URIs can be used as predicates, i.e., graph nodes can be used as edge labels (RDF has [meta-modeling abilities](#))
-

# Various types of literals

---

- (ex:thisLecture, ex:title, "graph databases")  
(untyped)
- (ex:thisLecture, ex:titlte, "graph databases"@en)  
(untyped, but assigned "English" (en) language)
- (ex:thisLecture, ex:titlte, "graph databases"^^xsd:string)  
(explicit type string)

Other types:

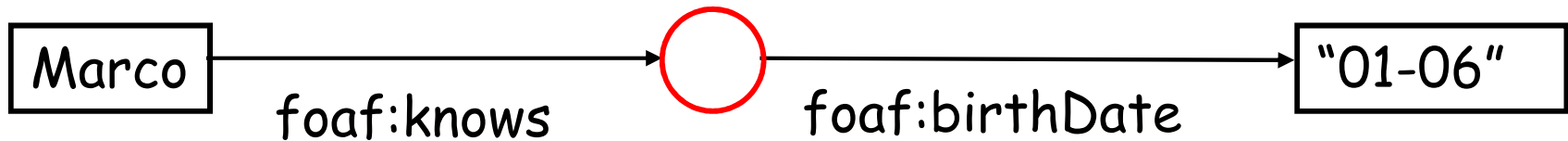
- xsd:decimal
  - xsd:integer
  - xsd:float
  - xsd:boolean
  - xsd:date
  - xsd:time
-

# 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

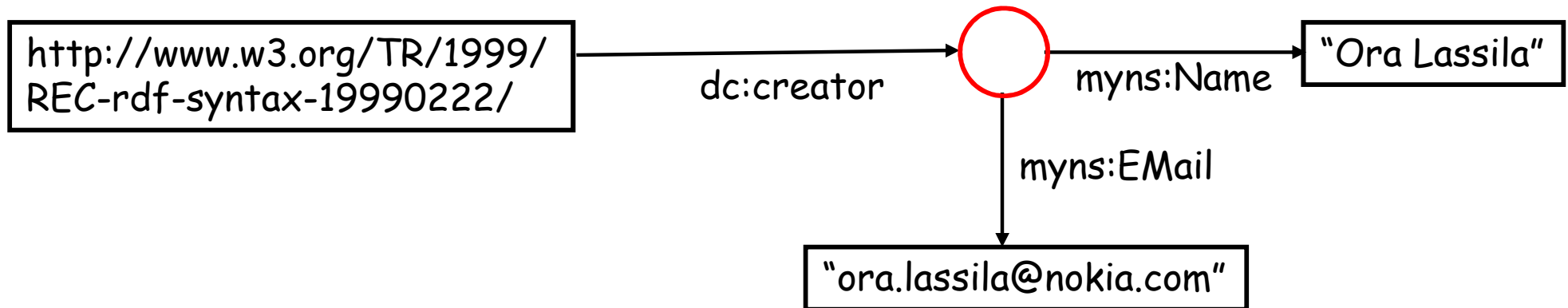


```
Marco    foaf:knows    _:X.  
_:X     foaf:birthDate "01-06".
```

# Blank nodes: unidentifiable resources

---

**Example:** The name of the creator of the specification of the RDF syntax is Ora Lassila and his email address is ora.lassila@nokia.com



```
http://www.w3.org/TR/1999/REC-rdf-syntax/19990222 dc:creator _:X.  
_:X myns:Name "Ora Lassila".  
_:X myns:Email "ora.lassila@nokia.com".
```

# RDF vocabulary

---

- RDF assigns a specific meaning to certain terms, the terms defined by the URI prefix <http://www.w3.org/1999/02/22-rdf-syntax-ns#> (usually abbreviated as [rdf:](#))
- **Some examples (meaning explained in the following slides)**
  - `rdf:type`
  - `rdf:Seq`, `rdf:Bag`, `rdf:Alt`
  - `rdf:_1`, `rdf:_2`, ..., `rdf:li`
  - `rdf:subject`
  - `rdf:predicate`
  - `rdf:object`
  - `rdf:Statement`
  - `rdf:Property`



# Containers

---

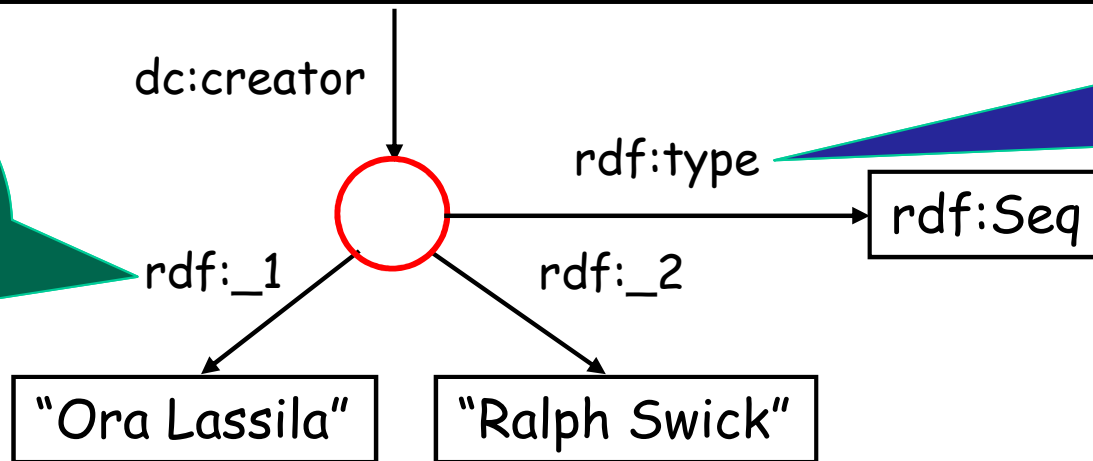
- **Containers** are collections
  - they allow grouping of resources (or literal values)
- Different types of containers exist
  - **bag** - unordered collection (`rdf:Bag`)
  - **seq** - ordered collection (= “sequence”) (`rdf:Seq`)
  - **alt** - represents alternatives (`rdf:Alt`)
- It is possible to express statements regarding the container (taken as a whole) or on its members
  - `rdf:_n` – n-th member of a sequence
  - `rdf:li` – element of a collection
- Duplicate values are permitted (no mechanism to enforce unique value constraints)

# Containers

**Example:** The names of the creators of the RDF syntax specification are (in order) Ora Lassila and Ralph Swick

<http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>

The object is the first element of the container occurring as subject



The subject is an instance of the class occurring as object

The class of ordered containers.

```
http://www.w3.org/TR/REC-rdf-syntax/ dc:creator _:X.  
_:X rdf:type rdf:Seq.  
_:X rdf:_1 "Ora Lassila".  
_:X rdf:_2 "Ralph Swick".
```

# 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
- ⇒ basic tool: reification, i.e., representation of an RDF assertion as a resource

# Reification

---

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

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

- “the New York Times claims that Joe is the author of book ABC”
- “the statement “The technical report on RDF was written by Ora Lassila” was written by the Library of Congress”

# 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 represent a triple through a blank node

# Reification: example

---

- the statement “The technical report on RDF was written by Ora Lassila” can be represented by the following four triples:

```
_:x rdf:predicate dc:creator.
```

```
_:x rdf:subject http://www.w3.org/TR/1999/REC-rdf-  
syntax-19990222/.
```

```
_:x rdf:object "Ora Lassila".
```

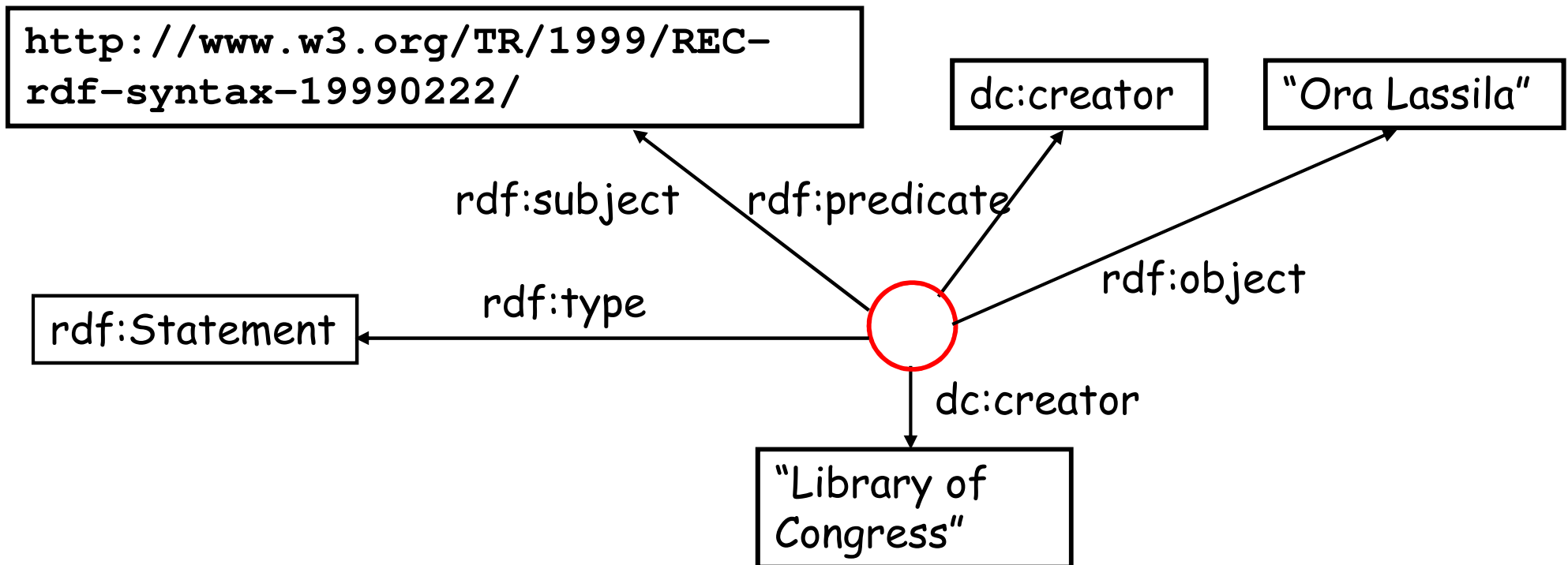
```
_:x rdf:type rdf:statement.
```

- The blank node `_:x` is the **reification** of the statement (it is an anonymous URI that represents the whole triple)
- Now, “The statement “The technical report on RDF was written by Ora Lassila” was written by the Library of Congress” can be represented using the bnode `_:x`, by adding to the above four triples the following triple:

```
_:x dc:creator "Library of Congress".
```

# Reification: example

The statement “The technical report on RDF was written by Ora Lassila” was written by the Library of Congress



# Exercise

---

Draw the RDF graph that represents the following assertions:

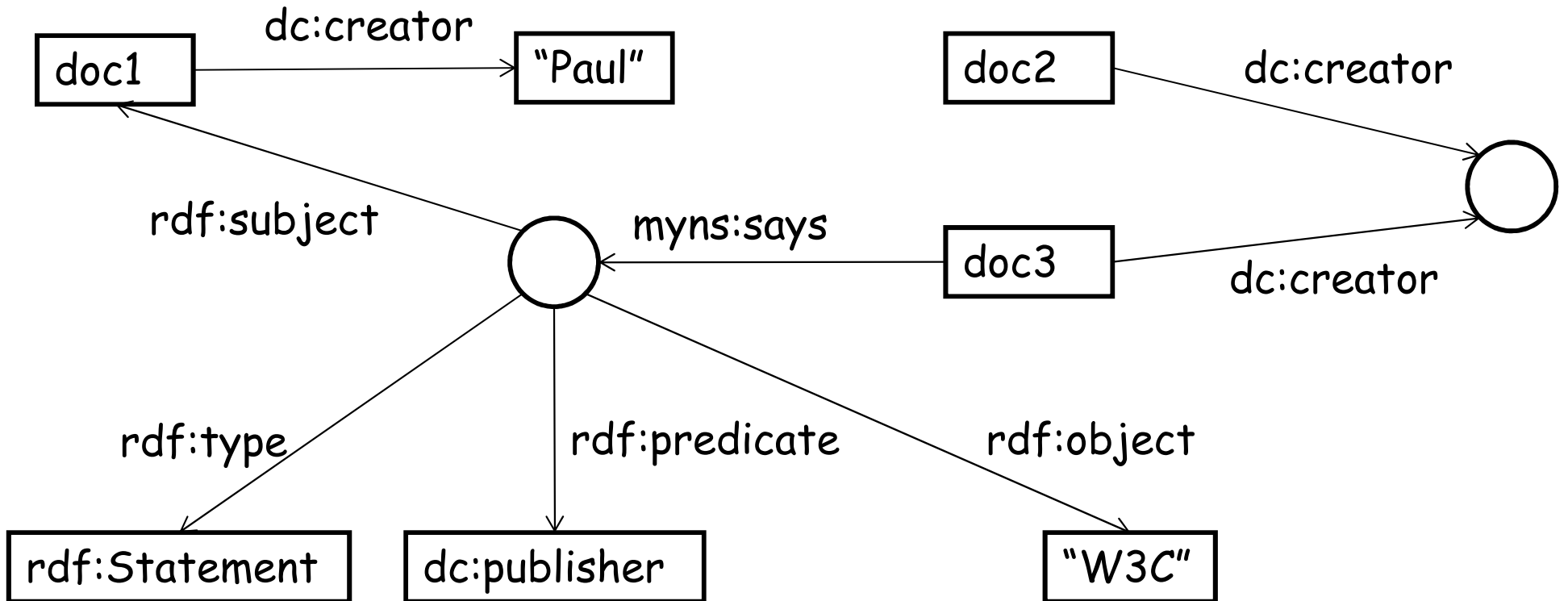
- Document 1 was created by Paul
- Document 2 and Document 3 were created by the same author (which is unknown)
- Document 3 says that Document 1 was published by the W3C

Use the predicates `dc:creator` and `dc:publisher`, and assume that the three documents are identified by the URIs `doc1`, `doc2`, and `doc3`, respectively.



# Solution

---



# 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
  - concrete (serialized) syntax: RDF/XML syntax

# RDF syntaxes

---

- N3 notation:  
`subject predicate object.`
- Turtle (Terse RDF Triple Language) notation. Example:  
`@prefix  
 rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.  
 :mary rdf:type <http://www.ex.org/Gardener>.  
 :mary :worksFor :ElJardinHaus.  
 :mary :name "Dalileh Jones"@en.  
 _:john :worksFor :ElJardinHas.  
 _:john :idNumber "54321"^^xsd:integer.`
- Concrete (serialized) syntax: RDF/XML syntax

# Turtle Notation: Example\*

---

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://example.org/stuff/1.0/> .

<http://www.w3.org/TR/rdf-syntax-grammar>
  dc:title "RDF/XML Syntax Specification (Revised)" ;
  ex:editor [
    ex:fullname "Dave Beckett";
    ex:homePage <http://purl.org/net/dajobe/>
  ] .
```

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

- The W3C technical report on RDF syntax and grammar, has the title RDF/XML Syntax Specification (Revised).
- That report's editor is a certain individual, who in turn
  - Has full name Dave Beckett.
  - Has a home page at <http://purl.org/net/dajobe/>.

---

\*Taken from [http://en.wikipedia.org/wiki/Turtle\\_\(syntax\)](http://en.wikipedia.org/wiki/Turtle_(syntax))

# Turtle Notation: Example\*

---

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

- The W3C technical report on RDF syntax and grammar, has the title RDF/XML Syntax Specification (Revised).
- That report's editor is a certain individual, who in turn
  - Has full name Dave Beckett.
  - Has a home page at <http://purl.org/net/dajobe/>.

*Here are the four triples of the RDF graph made explicit in N-Triples notation:*

```
<http://www.w3.org/TR/rdf-syntax-grammar> <http://purl.org/dc/elements/1.1/
  title> "RDF/XML Syntax Specification (Revised)" .
<http://www.w3.org/TR/rdf-syntax-grammar> <http://example.org/stuff/1.0/
  editor> _:bnode .
_:bnode <http://example.org/stuff/1.0/fullname> "Dave Beckett" .
_:bnode <http://example.org/stuff/1.0/homePage> <http://purl.org/net/dajobe> .
```

---

\*Taken from [http://en.wikipedia.org/wiki/Turtle\\_\(syntax\)](http://en.wikipedia.org/wiki/Turtle_(syntax))

# Turtle Notation: Example\*

---

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .
```

```
<#green-goblin>
  rel:enemyOf <#spiderman> ;
  a foaf:Person ;      # in the context of the Marvel universe
  foaf:name "Green Goblin" .
```

```
<#spiderman>
  rel:enemyOf <#green-goblin> ;
  a foaf:Person ;
  foaf:name "Spiderman", "Человек-паук"@ru .
```

# RDF/XML syntax

---

- A node in the RDF graph that represents a resource (labeled or not) is represented by an element `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 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 of type `rdf:Description`
- Values (literal) can be assigned with a type (the same defined in XML-Schema)

# RDF/XML syntax: Example

---

<http://www.w3.org/TR/REC-rdf-syntax/>

dc:creator

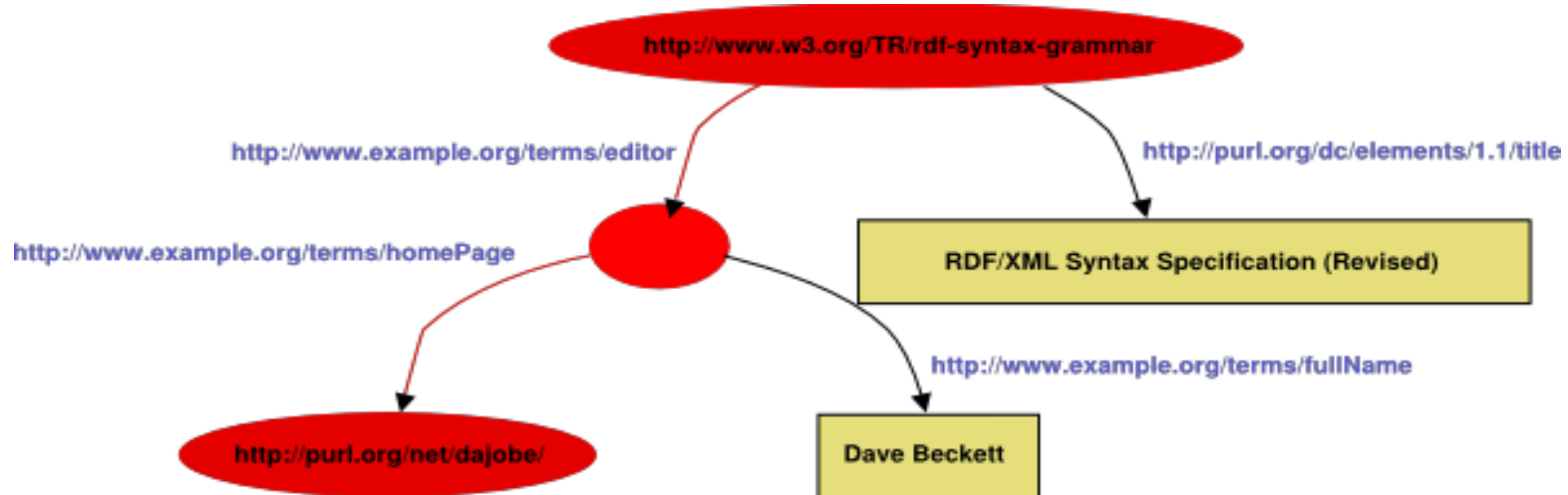
"Ora Lassila"

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://www.w3.org/TR/
    REC-rdf-syntax">
    <dc:creator>Ora Lassila</dc:creator>
  </rdf:Description>
</rdf:RDF>
```



# RDF/XML syntax: Example

---



```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
  <ex:editor>
    <rdf:Description>
      <ex:homePage>
        <rdf:Description rdf:about="http://purl.org/net/dajobe/">
          </rdf:Description>
        </ex:homePage>
        .....
      </rdf:Description>
    </ex:editor>
    .....
  </rdf:Description>
```

# RDF/XML syntax: simplifications

---

- A resource that is a literal and that is the object of a predicate may be encoded as the value of an attribute of the element that represents the subject. The type of such element is the label of the predicate
- The URI associated with a resource that is the object of a predicate and that is not the subject of any predicate can be encoded as the value of an attribute `rdf:resource` associated with the element that represents the predicate

# RDF/XML simplified 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/elements/1.1/">
```

```
<rdf:Description rdf:about="http://www.w3.org/TR/
  REC-rdf-syntax">
```

```
<dc:creator>Ora Lassila</dc:creator>
```

```
</rdf:Description>
```

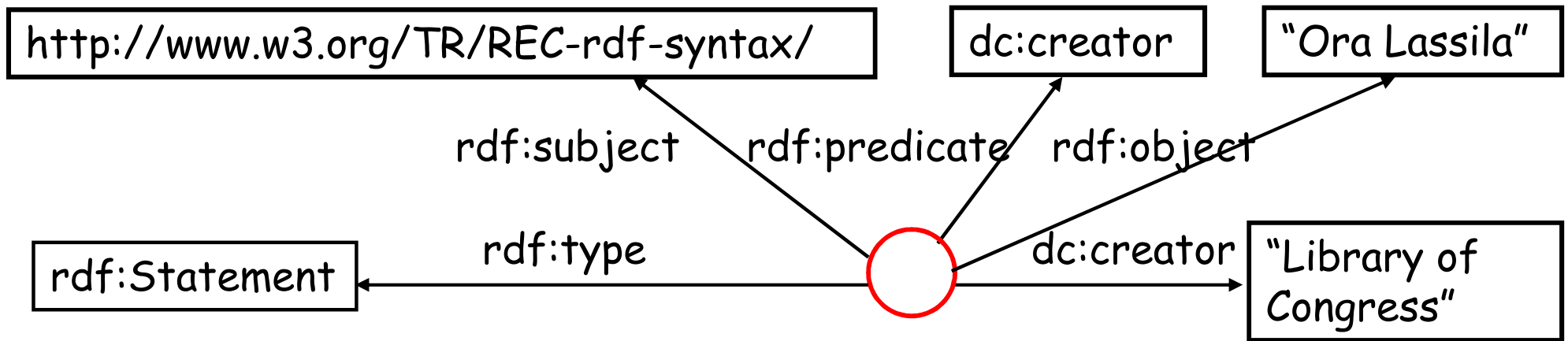
```
</rdf:RDF>
```



```
<rdf:Description rdf:about="http://www.w3.org/TR/
  REC-rdf-syntax" dc:creator="Ora Lassila"/>
```

# RDF/XML simplified syntax: Example

---

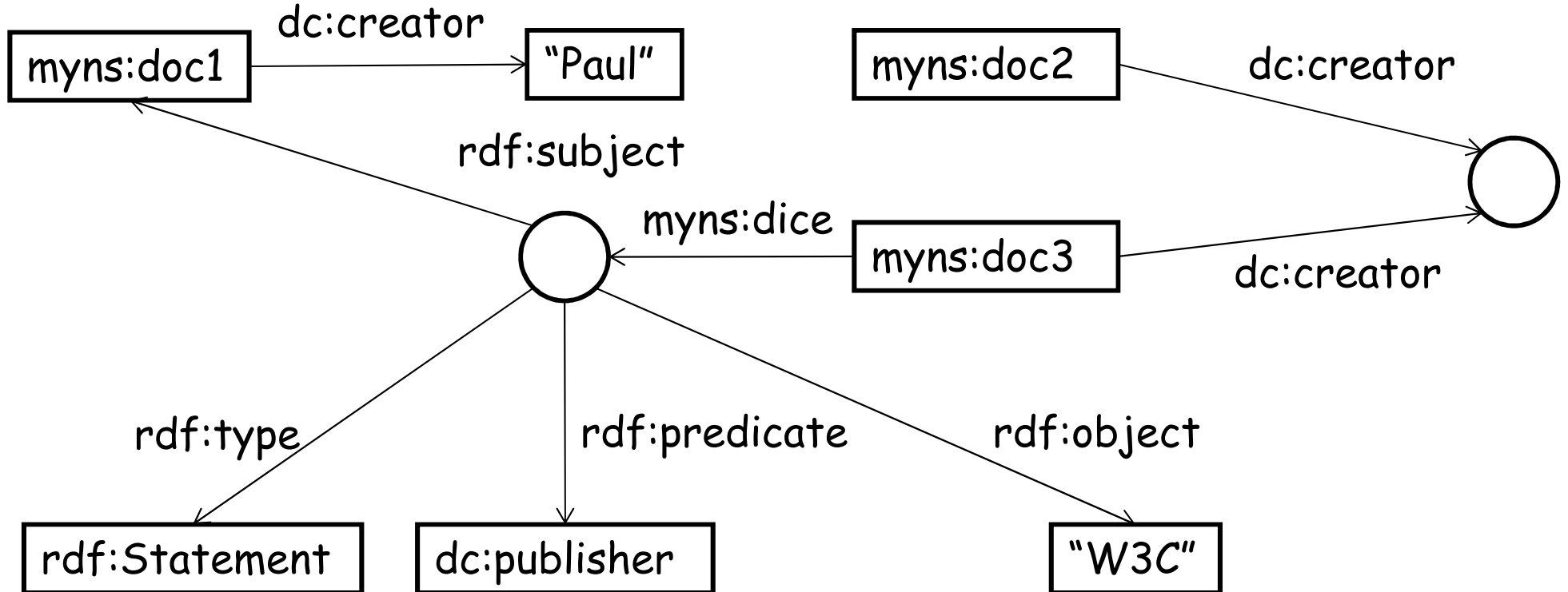


```
<rdf:Description dc:creator="Library of Congress"
  rdf:object="Ora Lassila" >
  <rdf:subject rdf:resource="http://www.w3.org/TR/
    REC-rdf-syntax"/>
  <rdf:predicate rdf:resource="dc:creator"/>
  <rdf:type rdf:resource="rdf:Statement"/>
</rdf:Description>
```

# Exercise 2: RDF/XML syntax

---

- Express the RDF graph of Exercise 1 through the RDF/XML syntax.



# Exercise 2: solution

---

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns="http://www.dis.uniroma1.it/~poggi/esempi_rdf/">
  <rdf:Description rdf:about="doc2">
    <dc:creator rdf:nodeID="C"/>
  </rdf:Description>
  <rdf:Description rdf:about="doc3">
    <dc:creator rdf:nodeID="C"/>
    <dice>
      <rdf:Description rdf:object="W3C">
        <rdf:type rdf:resource="http://www.w3.org/1999/02/22-
          rdf-syntax-ns#Statement"/>
        <rdf:predicate rdf:resource="http://purl.org/dc/
          elements/1.1/publisher"/>
        <rdf:subject>
          <rdf:Description rdf:about="doc1"
            dc:creator="Paul"/>
        </rdf:subject>
      </rdf:Description>
    </dice>
  </rdf:Description>
</rdf:RDF>
```

# RDF Schema

---

RDFS = RDF Schema

- Defines small **vocabulary** for RDF:
  - Class, subclassOf, type
  - Property, subPropertyOf
  - domain, range
- corresponds to a set of RDF predicates:
  - ⇒ meta-level
  - ⇒ special (predefined) “meaning”

# RDFS

---

- vocabulary for defining **classes** and **properties**
- vocabulary for classes:
  - **rdfs:Class** (a resource is a class)
  - **rdf:type**<sup>\*</sup> (a resource is an instance of a class)
  - **rdfs:subClassOf** (a resource is a subclass of another resource)

<sup>\*</sup>Already part of RDF built-in vocabulary (cf. the namespace rdf)



# RDFS

---

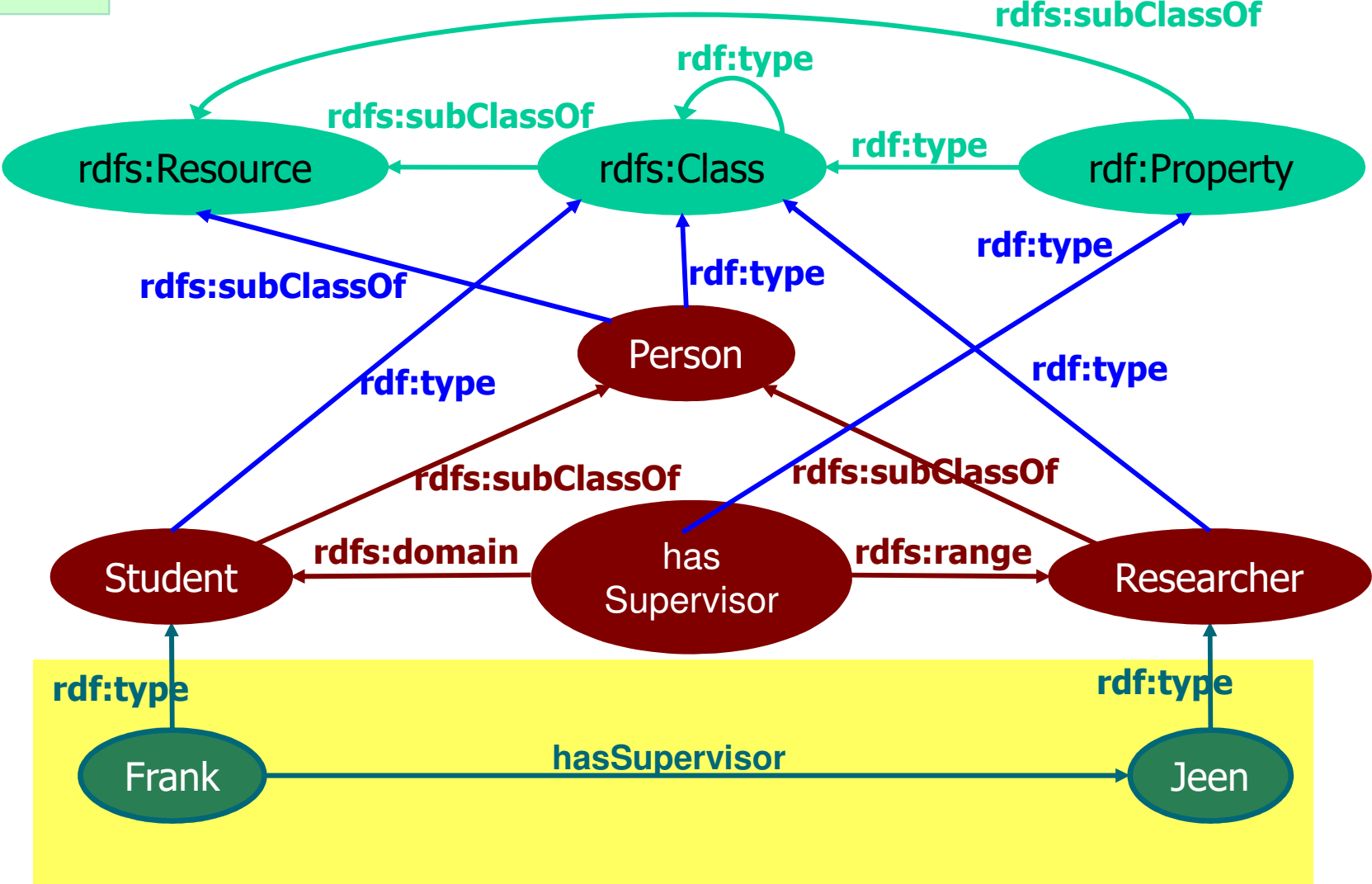
vocabulary for properties:

- **rdf:Property**\* (a resource is a property)
- **rdfs:domain** (denotes the first component of a property)
- **rdfs:range** (denotes the second component of a property)
- **rdfs:subPropertyOf** (expresses ISA between properties)

\*Already part of RDF built-in vocabulary (cf. the namespace rdf)

**Legenda**  
 RDF instance  
 RDFS schema  
 predefined in RDFS  
 logic. implied by the  
 RDFS semantics

# RDFS - example



# RDFS – example: triples

---

```
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 – example: XML syntax

---

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

  <rdf:Description rdf:about="#Student">
    <rdfs:subClassOf rdf:resource="#Person"/>
  </rdf:Description>

  <rdf:Description rdf:about="#Researcher">
    <rdfs:subClassOf rdf:resource="#Person"/>
  </rdf:Description>

  <rdf:Description rdf:about="#hasSupervisor">
    <rdfs:domain rdf:resource="#Student"/>
    <rdfs:range rdf:resource="#Researcher"/>
  </rdf:Description>

  <rdf:Description rdf:about="#Frank">
    <rdf:type rdf:resource="#Student"/>
    <hasSupervisor rdf:resource="#Jeen"/>
  </rdf:Description>

  <rdf:Description rdf:about="#Jeen">
    <rdf:type rdf:resource="#Researcher"/>
  </rdf:Description>

</rdf:RDF>
```

# RDFS

---

## example (classes):

```
(ex:MotorVehicle, rdf:type, rdfs:Class)
(ex:PassengerVehicle, rdf:type, rdfs:Class)
(ex:Van, rdf:type, rdfs:Class)
(ex:Truck, rdf:type, rdfs:Class)
(ex:MiniVan, rdf:type, rdfs:Class)
(ex:PassengerVehicle, rdfs:subClassOf,
  ex:MotorVehicle)
(ex:Van, rdfs:subClassOf, ex:MotorVehicle)
(ex:Truck, rdfs:subClassOf, ex:MotorVehicle)
(ex:MiniVan, rdfs:subClassOf, ex:Van)
(ex:MiniVan, rdfs:subClassOf, ex:PassengerVehicle)
```

# RDFS

example (classes):



# RDFS

---

example (properties):

`(ex:weight, rdf:type, rdf:Property)`

`(ex:weight, rdfs:domain, ex:MotorVehicle)`

`(ex:weight, rdfs:range, Integer)`

# RDFS: meta-modeling abilities

---

example (meta-classes):

```
(ex:MotorVehicle, rdf:type, rdfs:Class)
```

```
(ex:myClasses, rdf:type, rdfs:Class)
```

```
(ex:MotorVehicle, rdf:type, ex:myClasses)
```



# RDFS: XML syntax

---

example:

```
<rdf:Description rdf:about="MotorVehicle">  
  <rdf:type resource="http://www.w3.org/...#Class"/>  
  <rdfs:subClassOf rdf:resource="http://www.w3.org/...#Resource"/>  
</rdf:Description>
```

```
<rdf:Description rdf:about="Truck">  
  <rdf:type rdf:resource="http://www.w3.org/...#Class"/>  
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>  
</rdf:Description>
```

# RDFS: XML syntax

---

example (cont.):

```
<rdf:Description rdf:about="registeredTo">  
  <rdf:type resource="http://www.w3.org/...#Property"/>  
  <rdfs:domain rdf:resource="#MotorVehicle"/>  
  <rdfs:range rdf:resource="#Person"/>  
</rdf:Description>
```

```
<rdf:Description rdf:about="ownedBy">  
  <rdf:type resource="http://www.w3.org/...#Property"/>  
  <rdfs:subPropertyOf rdf:resource="#registeredTo"/>  
</rdf:Description>
```

# RDF + RDFS: semantics

---

- what is the exact meaning of an RDF(S) graph?
  - initially, a formal semantics was not defined!
  - main problems:
    - bnodes
    - meta-modeling
    - formal semantics for RDFS vocabulary
  - recently, a model-theoretic semantics has been provided
- ⇒ formal definition of entailment and query answering over RDF(S) graphs

# Incomplete information in RDF graphs

---

- bnodes = existential values (null values)  
⇒ introduce incomplete information in RDF graphs
- an RDF graph can be seen as an **incomplete database** represented in the form of a **naïve table**, i.e., relational tables containing values and named existential variable (also called labeled nulls)
- an RDF graph can be thus represented by a unique (naïve) table T, with values being constants or named existential variables

# RDF + RDFS: semantics

---

problems with meta-data:

`(#a rdf:type #C)`

`(#C rdf:type #R)`

`(#R rdf:type #a)`

or

`(#C rdf:type #C)`

are correct (formally meaningful) RDF statements

⇒ but no intuitive semantics

# Formal semantics for RDF + RDFS

---

- formal semantics for RDFS vocabulary
- RDFS statements = **constraints** over the RDF graph
- entailment in RDF + RDFS = reasoning (query answering) over an **incomplete database with constraints**

# Exercise 3: RDF/RDFS model

---

- Draw the graph representing the following assertions
  - URI1 and URI2 are classes
  - URI3 is a property
  - URI4 is an instance of the class URI1
  - URI5 and URI6 are instances of the class URI2
  - URI3 has URI1 as domain and URI2 as range
  - (URI4, URI6) is an instance of the property URI3
- Express the graph in XML syntax.

## Exercise 3: solution (triples)

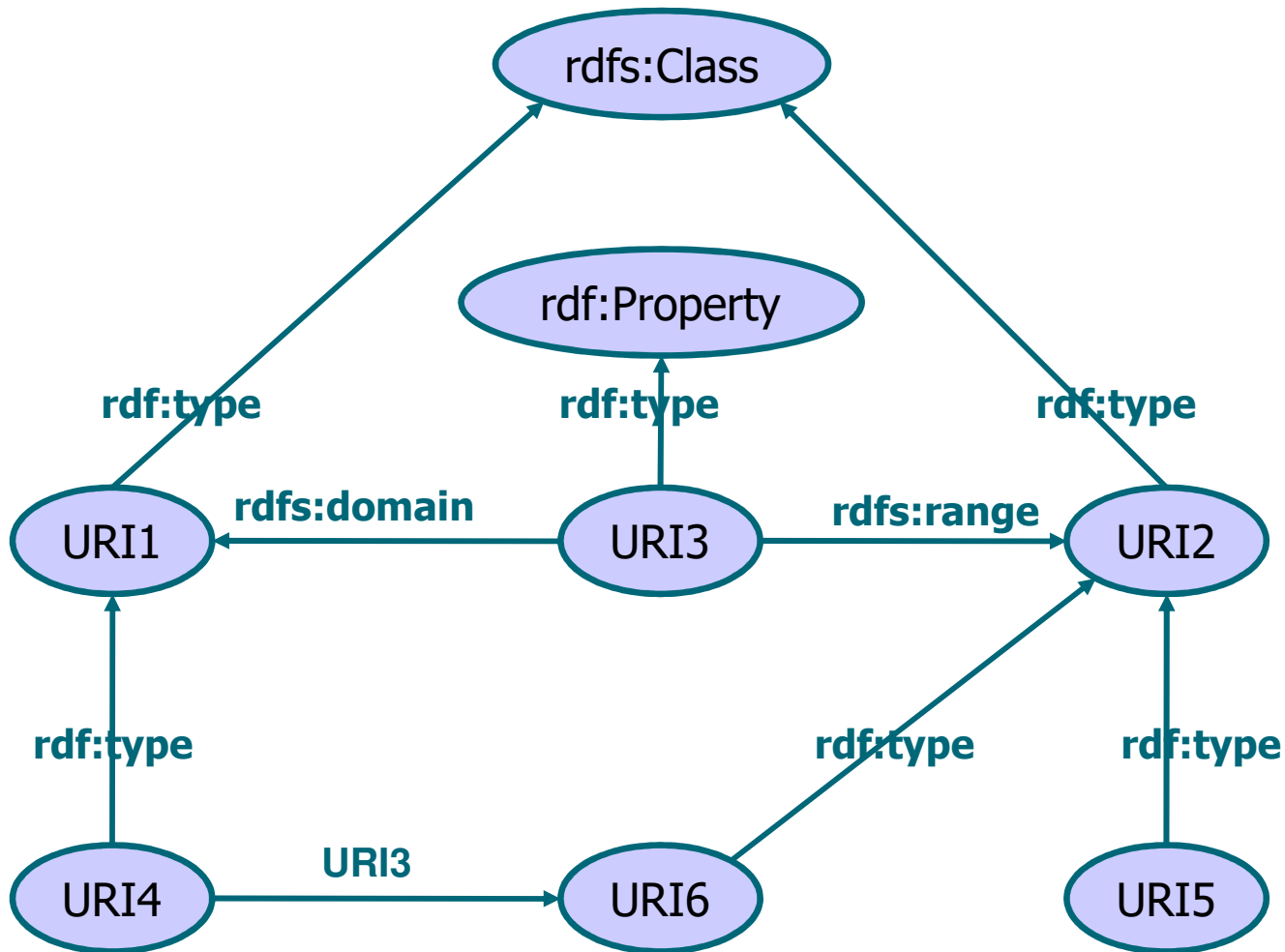
---

```
URI1  rdf:type  rdfs:Class.  
URI2  rdf:type  rdfs:Class.  
URI3  rdf:type  rdf:Property.  
URI4  rdf:type  URI1.  
URI5  rdf:type  URI2.  
URI6  rdf:type  URI2.  
URI3  rdfs:domain  URI1.  
URI3  rdfs:range   URI2.  
URI4  URI3  URI6.
```



# Exercise 3: solution (graph)

---



# Exercise 3: solution (XML syntax)

---

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

  <rdf:Description rdf:about="URI1">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>

  <rdf:Description rdf:about="URI2">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>

  <rdf:Description rdf:about="URI3">
    <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
    <rdfs:domain rdf:resource="URI1"/>
    <rdfs:range rdf:resource="URI2"/>
  </rdf:Description>

  <rdf:Description rdf:about="URI4">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
    <URI3 rdf:resource="URI6"/>
  </rdf:Description>

  <rdf:Description rdf:about="URI5">
    <rdf:type rdf:resource="URI2"/>
  </rdf:Description>

  <rdf:Description rdf:about="URI6">
    <rdf:type rdf:resource="URI2"/>
  </rdf:Description>

</rdf:RDF>
```

# Graph Databases

- Introduction to Graph Databases
- Resource Description Framework (RDF)
- Querying RDF databases: The SPARQL language
- RDF storage
- Linked data
- Tools

# 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)
  - **dataset definitions**, to specify the graph to be queried (they can be more than one)
  - **SELECT clause**, to specify the information to be returned
  - **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: <...>
...
# dataset definition
FROM <...>
# data to be returned
SELECT ...
# 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 result tuples;
  - 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 – example

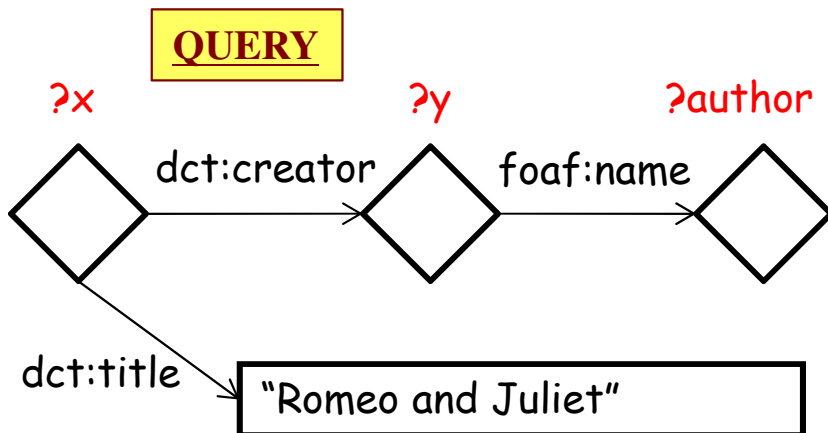
---

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

- Variables are outlined through the "?" prefix ("\$" is also possible).
- The ?author variable will be returned as result.
- The FROM clause specifies the URI of the graph to be queried
- The SPARQL query processor returns all hits matching the pattern of the four RDF-triples.
- "property orientation" (class matches can be conducted solely through class-attributes/properties)

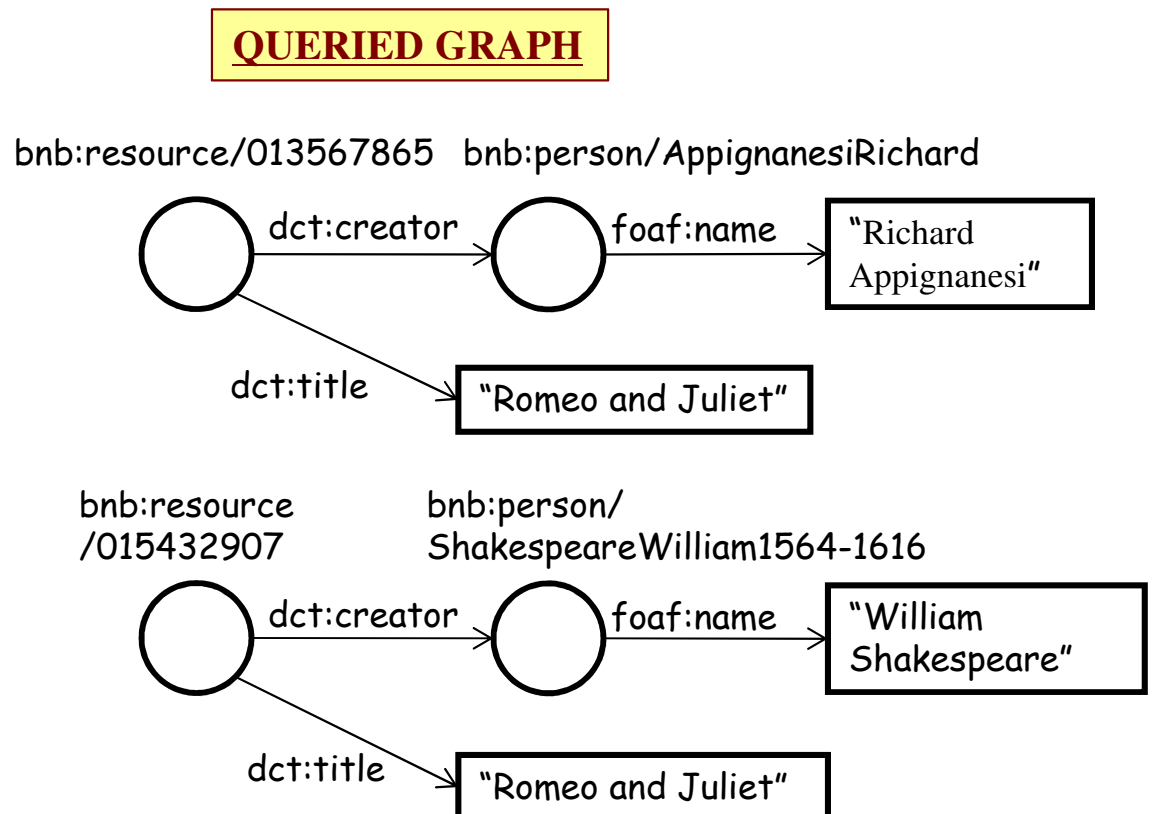
# SPARQL – query evaluation

The query returns all resources R for which there are resources X, Y, such that replacing variables ?authors, ?x and ?y, respectively, you get the triples in the queried graph.



**RESULT**

author
"Richard Appignanesi"
"William Shakespeare"





# SPARQL endpoints

---

- SPARQL queries are performed on **RDF dataset** (i.e, graphs)
  - A **SPARQL endpoint** accepts queries and returns results via the HTTP protocol
    - **generic endpoints** query all RDF datasets datasets that are accessible via the Web
      - <http://semantic.ckan.net/sparql>,  
<http://lod.openlinksw.com/sparql>
    - **Dedicated endpoints** are intended to query one or more specific dataset
      - <http://bnb.data.bl.uk/sparql>,  
<http://dbpedia.org/sparql> ...
  - The FROM clause, in principle, is mandatory, but
    - when the endpoint is dedicated, typically, 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
  - XML
  - HTML
  - Notation3
  - ...
- As an example, look at the generic SPARQL endpoint `http://lod.openlinksw.com/sparql`

# SPARQL query – 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>

# SPARQL – use of filters: 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") }
```

result:

"Johnny Lee Outlaw"	<mailto:jlow@example.com>
---------------------	---------------------------

# Predicates that can be used in the FILTER clause

---

- Logical connectives:
  - ! (NOT)
  - && (AND)
  - || (OR)
- Comparison: >, <, =, != (not equal), IN, NOT IN,..
- Test: isURI, isBlank, isLiteral, isNumeric, ...
- ...

# SPARQL – example of query on DBPedia

---

- Return the worldwide landlocked countries with more than 15 millions of inhabitants

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbpedia.org/class/yago/>
PREFIX prop: <http://dbpedia.org/property/>
SELECT ?country_name ?population
WHERE {
    ?country rdf:type type:LandlockedCountries.
    ?country rdfs:label ?country_name.
    ?country prop:populationEstimate ?population .
    FILTER (?population > 15000000)}
```

- Execute the query on the DBPedia endpoint  
(<http://dbpedia.org/snorql/>)

# SPARQL – optional patterns: example 1

---

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>

# SPARQL – optional patterns: example 2

---

```
@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> .
_:b rdf:type foaf:Person .
_:b foaf:name "Bob" .
```

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

"Alice"	<mailto:alice@example.com>
"Alice"	<mailto:alice@work.example>
"Bob"	



# SPARQL – optional patterns: example 3

---

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

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

- Run the query on the generic endpoint of OpenLink Software (<http://lod.openlinksw.com/sparql>) and compare the results obtained with those returned by the version of the query in which ?i10 and ?i13 are not optional.

# SPARQL – UNIONs of graph patterns

---

A graph pattern can be defined as the union of two (or more) 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
from <http://thedatahub.org/dataset/bluk-bnb>
WHERE {{?x dct:title "Romeo and Juliet".
        ?x bibo:isbn10 ?i} UNION
        {?x dct:title "Romeo and Juliet".
        ?x bibo:isbn13 ?i}}
```

# 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>?`

```
select distinct ?p
from <http://thedatahub.org/dataset/bluk-bnb>
where {<http://bnb.data.bl.uk/id/resource/
                                015432907> ?p ?v}
```

# Exercise 4: SPARQL queries

---

Express through a SPARQL query the following request:

- Return the URIs that have both an author and a creation date (you may use `dc:creator` and `dc:date` as predicates)

# Exercise 4: solution

---

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?x
WHERE { ?x dc:creator ?y .
        ?x dc:date ?z . }
```

# Exercise 5: SPARQL queries

---

Express through a SPARQL query the following request:

- Return those predicates that have both `myns:URI1` and `myns:URI2` as subjects, where `myns` is the name space `"http://www.dis.uniroma1.it/~poggi/esempi_rdf/"`

# Exercise 5: solution

---

```
PREFIX myns :  
  <http://www.dis.uniroma1.it/~poggi/esempi_rdf/>  
SELECT ?x  
WHERE { myns:uri1 ?x ?y .  
        myns:uri2 ?x ?z . }
```

# Exercise 6: SPARQL queries

---

Express through a SPARQL query the following request:

- Return those predicates that have either `myns:URI1` or `myns:URI2` as subjects, where `myns` is the name space `"http://www.dis.uniroma1.it/~poggi/esempi_rdf/"`



# Exercise 6: solution

---

```
PREFIX myns:
```

```
  <http://www.dis.uniroma1.it/~poggi/esempi_rdf/>
```

```
SELECT ?x
```

```
WHERE { { myns:uri1 ?x ?y } UNION  
        { myns:uri2 ?x ?z } }
```

# Exercise 7: SPARQL queries

---

Express through a SPARQL query the following request:

- Return the name of the authors of resources that have a creation date (you may use the predicates `dc:creator`, `dc:date`, and `foaf:name`)

# Exercise 7: solution

---

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?z
WHERE {
  ?x dc:creator ?y .
      ?y foaf:name ?z .
  ?x dc:date ?w }
```

# Exercise 8: SPARQL queries

---

Express through a SPARQL query the following request:

- Return the names of the authors and the creation date of resources that have an author (who has a name) and possibly have a creation date.

# Exercise 8: solution

---

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?z ?w
WHERE {
  ?x dc:creator ?y .
  ?y foaf:name ?z .
  OPTIONAL { ?x dc:date ?w } }
```

# SPARQL 1.1: property paths

---

- SPARQL 1.1 (released in 2013) extends the first version of SPARQL in different ways (e.g., aggregate functions, entailment regimes)
- In particular, it allows for expressing **property paths** in queries
- A property path is essentially a regular expression using properties (URIs)
- Property paths allow for expressing paths of arbitrary length along the RDF graph, thus providing a fundamental graph-oriented feature to SPARQL

# Property paths in SPARQL

Syntax Form	Property Path Expr. Name	Matches
iri	PredicatePath	An IRI. A path of length one.
^elt	InversePath	Inverse path (object to subject).
elt1 / elt2	SequencePath	A sequence path of elt1 followed by elt2.
elt1   elt2	AlternativePath	A alternative path of elt1 or elt2 (all possibilities are tried).
elt*	ZeroOrMorePath	A path that connects the subject and object of the path by zero or more matches of elt.
elt+	OneOrMorePath	A path that connects the subject and object of the path by one or more matches of elt.
elt?	ZeroOrOnePath	A path that connects the subject and object of the path by zero or one matches of elt.
!iri or !(iri <sub>1</sub>   ...   iri <sub>n</sub> )	NegatedPropertySet	Negated property set. An IRI which is not one of iri <sub>i</sub> . !iri is short for !(iri).
!^iri or !(^iri <sub>1</sub>   ...   ^iri <sub>n</sub> )	NegatedPropertySet	Negated property set where the excluded matches are based on reversed path. That is, not one of iri <sub>1</sub> ...iri <sub>n</sub> as reverse paths. !^iri is short for !(^iri).
!(iri <sub>1</sub>   ...   iri <sub>j</sub>   ^iri <sub>j+1</sub>   ...   ^iri <sub>n</sub> )	NegatedPropertySet	A combination of forward and reverse properties in a negated property set.
(elt)		A group path elt, brackets control precedence.

# Exercise 9: property paths

---

Express through a SPARQL query the following request:

- Return the names of all the ancestors and the descendants of John, assuming that the RDF graph uses the properties `:hasFather` and `:hasMother` to express kinship relations.



# Exercise 9: solution

---

```
PREFIX ex: <http://example.org/example/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?z
WHERE { ex:John (ex:hasFather | ex:hasMother)+ ?x
        .
        ?x foaf:name ?z . }
UNION
        { ?x (ex:hasFather | ex:hasMother)+
          ex:John .
          ?x foaf:name ?z . }
```

# Exercise 10: property paths

---

Express through a SPARQL query the following request:

- Return all the classes which John belongs to.

# Exercise 10: solution

---

```
PREFIX ex: <http://example.org/example/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
  schema#>
SELECT ?z
WHERE { ex:John rdf:type ?c .
        ?c rdfs:subClassOf* ?x . }
```

Or, equivalently:

```
PREFIX ex: <http://example.org/example/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
  schema#>
SELECT ?z
WHERE { ex:John rdf:type/rdfs:subclassOf* ?c . }
```

# Graph Databases

- Introduction to Graph Databases
- Resource Description Framework (RDF)
- Querying RDF databases: The SPARQL language
- **RDF storage**
- Linked data
- Tools

# 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 RDF datasets rather than single RDF graphs. 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.
- So-called **Unclustered BTree indexes** are four different sets of BTree indexes on the triple table (s,p,o):
  - 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**, instead, provide various sorted versions of the triple store table according to various permutation of the sequence s,p,o, allowing fast access to clusters of trees

# The relational perspective – Horizontal representation

---

- A different approach under the relational perspective provides an **horizontal representation** of RDF
- According to such representation, data is conceptually stored in a single table that has **one column for each predicate value that occurs in the RDF graph, and one row for each subject value**. For each (s,p,o) triple, the object o is placed in the p column of row s.



# The horizontal representation - example

---

*rdf triple*

```
{ <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 >, ... }
```

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

*relational  
horizontal  
representation*

# The horizontal representation

---

- As can be seen from the previous example, it is rare 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 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, a different table 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 (supports 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 essentially corresponds to a triple.
- Note that, hence, both the empty cell issue and the multiple object issue are solved at the same time.

# The relational perspective – storage of URIs and literals

---

- Independently 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:

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

# 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 from the **information retrieval literature** can then be specialized to support queries patterns that retrieve entities based on particular attributes and/or values
- For example, in the previous representation we have that user8604 is retrieved when searching for entities born in 1975 (i.e., have 1975 as a value on attribute birthdate) as well as when searching for entities with friends who like Impressionist music. Note that entity user3789 is not retrieved by either of these queries.
- Specific tools provide peculiar solutions to these problems



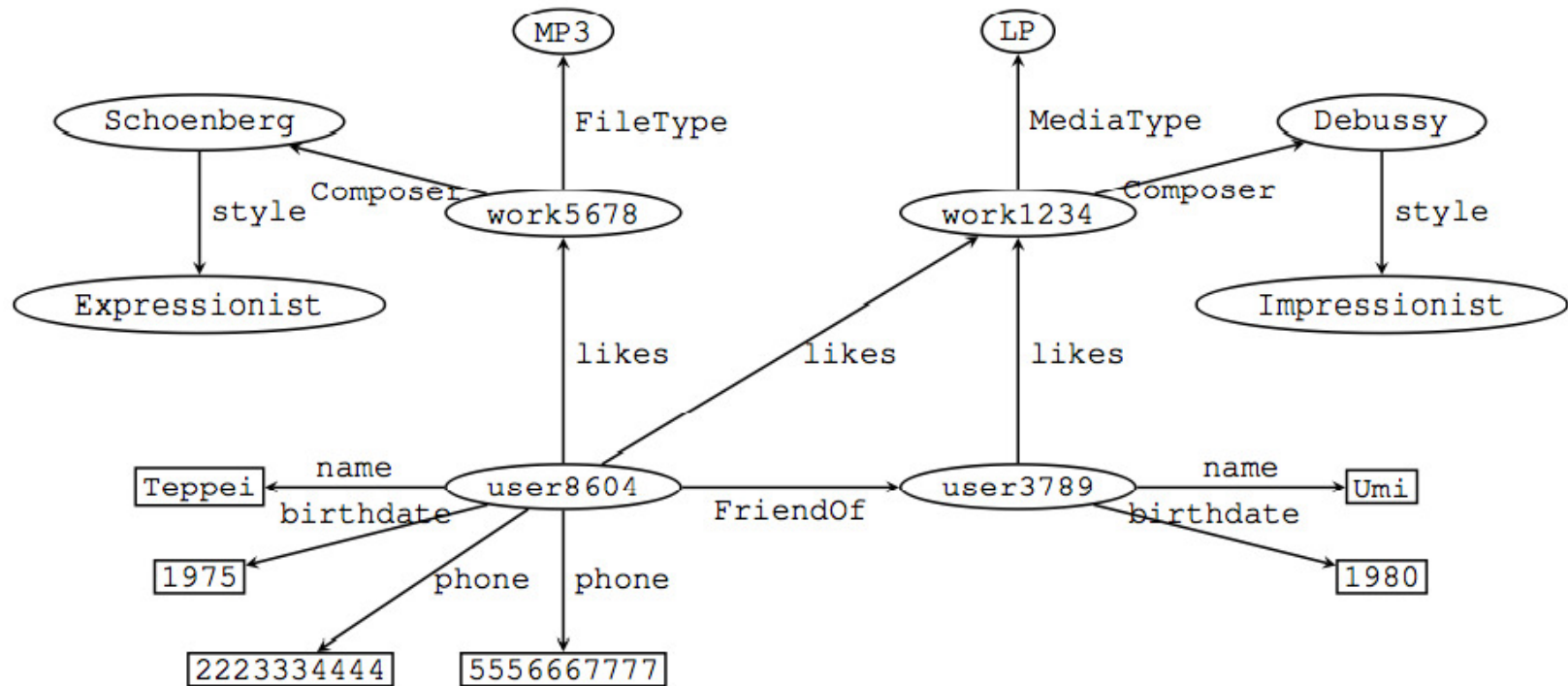
# The graph-based perspective for storing RDF graphs

---

- 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 query patterns supported in this perspective are graph-theoretic queries such as reachability between 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

---



# Graph Databases

- Introduction to Graph Databases
- Resource Description Framework (RDF)
- Querying RDF databases: The SPARQL language
- RDF storage
- **Linked data**
- Tools

# RDF in the real world

---

- RDF and SPARQL are W3C standards
- Widespread use for metadata representation, e.g.
  - Meta Content Framework (MCF) developed by Apple as a specification of a content format for structuring metadata about web sites and other data (it is specified in a language which is a sort of ancestor of RDF)
  - Adobe XMP (Extensible Metadata Platform), an RDF based schema that offers properties that provide basic descriptive information on files
- Oracle supports RDF, and provides an extension of SQL to query RDF data
- HP has a big lab (in Bristol) developing specialized data stores for RDF (it also initiated the development of the Jena framework for RDF graph management, carried out until october 2009 – then by Apache Software foundation )<sub>148</sub>

# RDF in the real world

---

- current main application of RDF: **linked data**
- linked data = using the Web to create **typed links** between data from different sources
- i.e.: create a **Web of data**
- DBpedia, Geonames, US Census, EuroStat, MusicBrainz, BBC Programmes, Flickr, DBLP, PubMed, UniProt, FOAF, SIOC, OpenCyc, UMBEL, Virtual Observatories, freebase,...
- each source: up to several million triples
- overall: over 31 billions triples (2012)

# Linked Data

---

**Linked Data:** set of best practices for publishing and connecting structured data on the Web using URIs and RDF

Basic idea: apply the general architecture of the World Wide Web to the task of sharing structured data on global scale

- The Web is built on the idea of setting hyperlinks between documents that may reside on different Web servers.
- It is built on a small set of simple standards:
  - Global identification mechanism: URIs, IRIs
  - Univeral access mechanism: HTTP
  - Standardized content format: HTML

Linked Data builds directly on Web architecture and applies this architecture to the task of sharing data on global scale

# Linked data principles

---

1. Use **URIs** as names for things.
2. Use HTTP URIs, so that people can look up those names (**dereferenceable URIs**).
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include **links** to other URIs, so that they can discover more things.

**Dereferenceability** = URIs are not just used for identifying entities: since they can be used in the same way as URLs, they also enable locating and retrieving resources describing and representing these entities on the Web.

# Linked data principles

---

Just as hyperlinks in the classic Web connect documents into a single global information space, Linked Data uses hyperlinks to connect disparate data into a single **global data space**.

These links, in turn, enable applications to navigate the data space.

For example, a Linked Data application that has looked up a URI and retrieved RDF data describing a person may follow links from that data to data on different Web servers, describing, for instance, the place where the person lives or the company for which the person works.



# Linked Data lifecycle

---



1. Extraction
2. Storage & Querying
3. Authoring
4. Linking
5. Enrichment
6. Quality Analysis
7. Evolution & Repair
8. Search, Browsing & Exploration

# Linked Data lifecycle

---

- **Extraction:** Map Non-RDF data into the RDF format
- **Storage & Querying:** Once there is a critical mass of RDF data, mechanisms have to be in place to store, index and query this RDF data efficiently.
- **Authoring:** Users must have the opportunity to create new structured information or to correct and extend existing ones
- **Linking:** Links between related entities have to be established (possibly applying schema matching and record linkage techniques).
- **Enrichment:** Raw RDF data should be enriched with higher level structures (e.g., schema information)

# Linked Data lifecycle

---

- **Quality Analysis:** As with the Document Web, the Data Web contains a variety of information of different quality. Hence, it is important to devise strategies for assessing the quality of data published on the Data Web
- **Evolution & Repair:** Once problems are detected, strategies for repairing these problems and supporting the evolution of Linked Data are required.
- **Search, Browsing & Exploration:** users have to be empowered to browse, search and explore the information available on the Data Web in a fast and user friendly manner.

# Open/Closed Linked Data

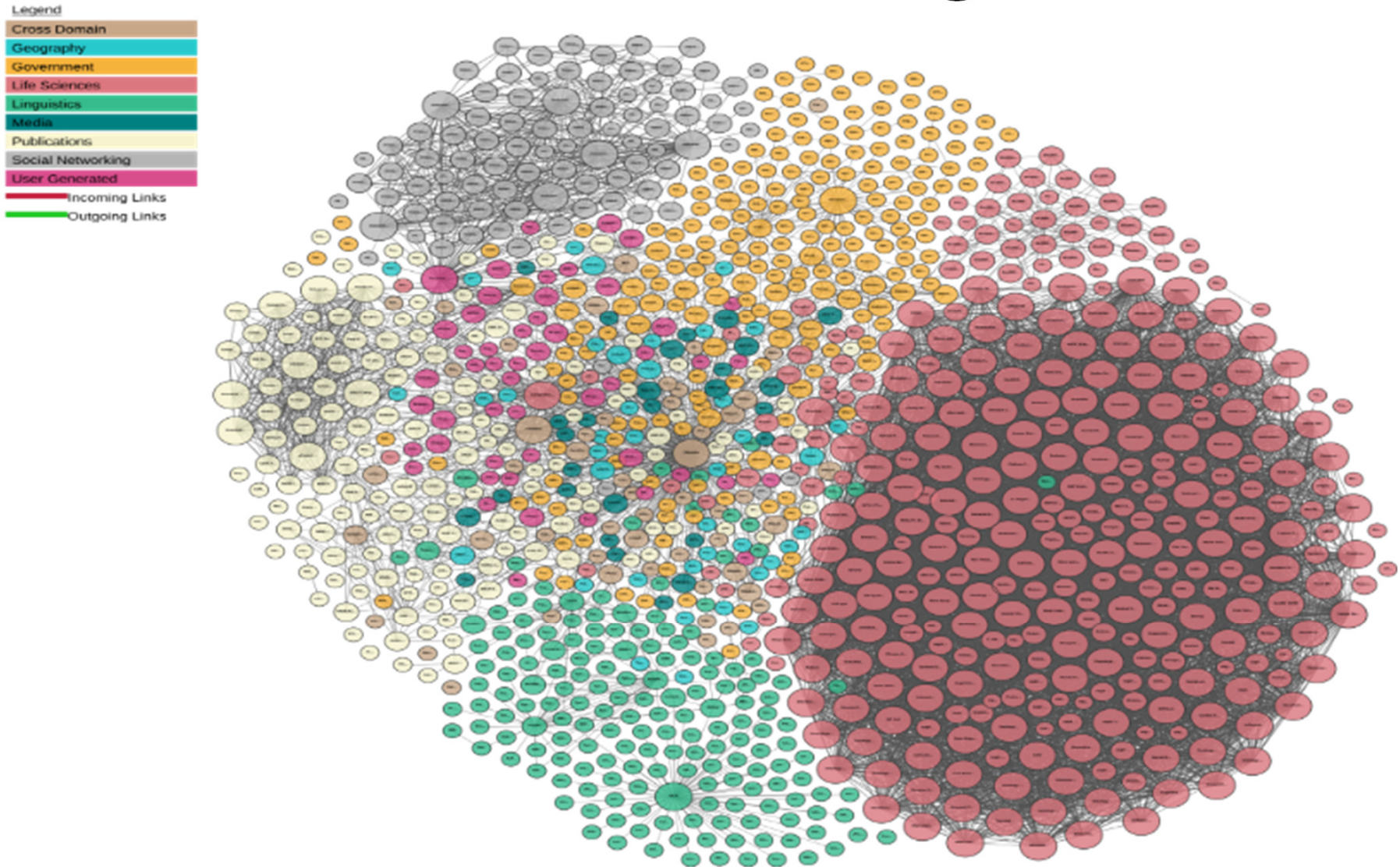
---

Linked data may be **open** (publicly accessible and reusable) or **closed**

**Linking Open Data (LOD)**: project which aims at creating an open Linked Data network

<http://lod-cloud.net>

# The LOD cloud diagram



Linking Open Data cloud diagram 2017, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak. <http://lod-cloud.net/>

# Use of RDF vocabularies

---

- Crucial aspect of Linked Data (and of RDF usage in general): which URIs represent predicates (links)?
- Recommended practice in LOD: if possible, **use existing RDF vocabularies** (and preferably the most popular ones)
- In this way, a de-facto standard is created: all LOD sites use the same URI to represent the same property, and the semantics of such properties is shared (i.e., known by every application)
- This makes it possible for all applications to really understand the semantics of links

# Popular vocabularies

---

- **Friend-of-a-Friend (FOAF)**, vocabulary for describing people
- **Dublin Core (DC)** defines general metadata attributes
- **Semantically-Interlinked Online Communities (SIOC)**, vocabulary for representing online communities
- **Description of a Project (DOAP)**, vocabulary for describing projects
- **Simple Knowledge Organization System (SKOS)**, vocabulary for representing taxonomies and loosely structured knowledge
- **Music Ontology** provides terms for describing artists, albums and tracks
- **Review Vocabulary**, vocabulary for representing reviews
- **Creative Commons (CC)**, vocabulary for describing license terms

# Graph Databases

- Introduction to Graph Databases
- Resource Description Framework (RDF)
- Querying RDF databases: The SPARQL language
- RDF storage
- Linked data
- **Tools**



# RDF/SPARQL tools

---

- **Jena** = Java framework for handling RDF models and SPARQL queries (<http://jena.sourceforge.net/>)
- **ARC** = PHP implementation of a RDF/SPARQL engine (<http://arc.semsol.org/>)
- **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/>)

# RDF/SPARQL (and graph database) tools

---

- **Allegrograph**

(<http://www.franz.com/agraph/allegrograph/>) = it is a native triple store providing support for SPARQL queries over RDF datasets. It also provide Prolog query APIs and offers built-in reasoner over RDFS++ (i.e., RDFS predicates plus some of OWL predicates, the W3C Web Ontology language).

- ...and many more, see <http://esw.w3.org/topic/SparqlImplementations>

# RDF/SPARQL (and graph database) tools

---

- RDF datasets can be also stored in non-native RDF storage systems
- Graph databases (as Allegrograph) are the most suited ones: e.g., the most widely used graph database today, Neo4j (<http://www.neo4j.org/>), provides an RDF/SPARQL module which relies on a native storage of data in the form of property graph databases
- Other kinds of NoSQL databases can be used to store RDF triples, and often provide some kind of SPARQL query support (e.g, HBase, Couchbase, Cassandra).
- In all these cases, however, no specific index mechanisms for RDF datasets are guaranteed.
- For some deepenings on the use of NoSQL databases for RDF storage, we refer to Cudré-Mauroux et al. NoSQL Databases for RDF: An Empirical Evaluation. ISWC 2013.