Data Management for Data Science

*Master of Science in Data Science*
*Facoltà di Ing. dell'Informazione, Informatica e Statistica*
*Sapienza Università di Roma*

AA 2018/2019

# Graph Databases

**Domenico Lembo**
*Dipartimento di Ingegneria Informatica,*
*Automatica e Gestionale A. Ruberti*

# References

- 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 28 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.
- The slides on RDF and SPARQL are adapted from slides originally produced by prof. Riccardo Rosati, Sapienza University. Some examples are by Antonella Poggi, Sapienza University.
- 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)
- RDF storage
- Querying RDF databases: The SPARQL language
- 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 are bigger in volume, change more rapidly, and are more structurally varied (in a definition, *Big Data*) than those that can be dealt with by traditional RDBMS deployments.
- The NOSQL movement has arisen in response to these challenges.

# Limits of relation 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 substainable when we require sub-second response to queries.

# 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.

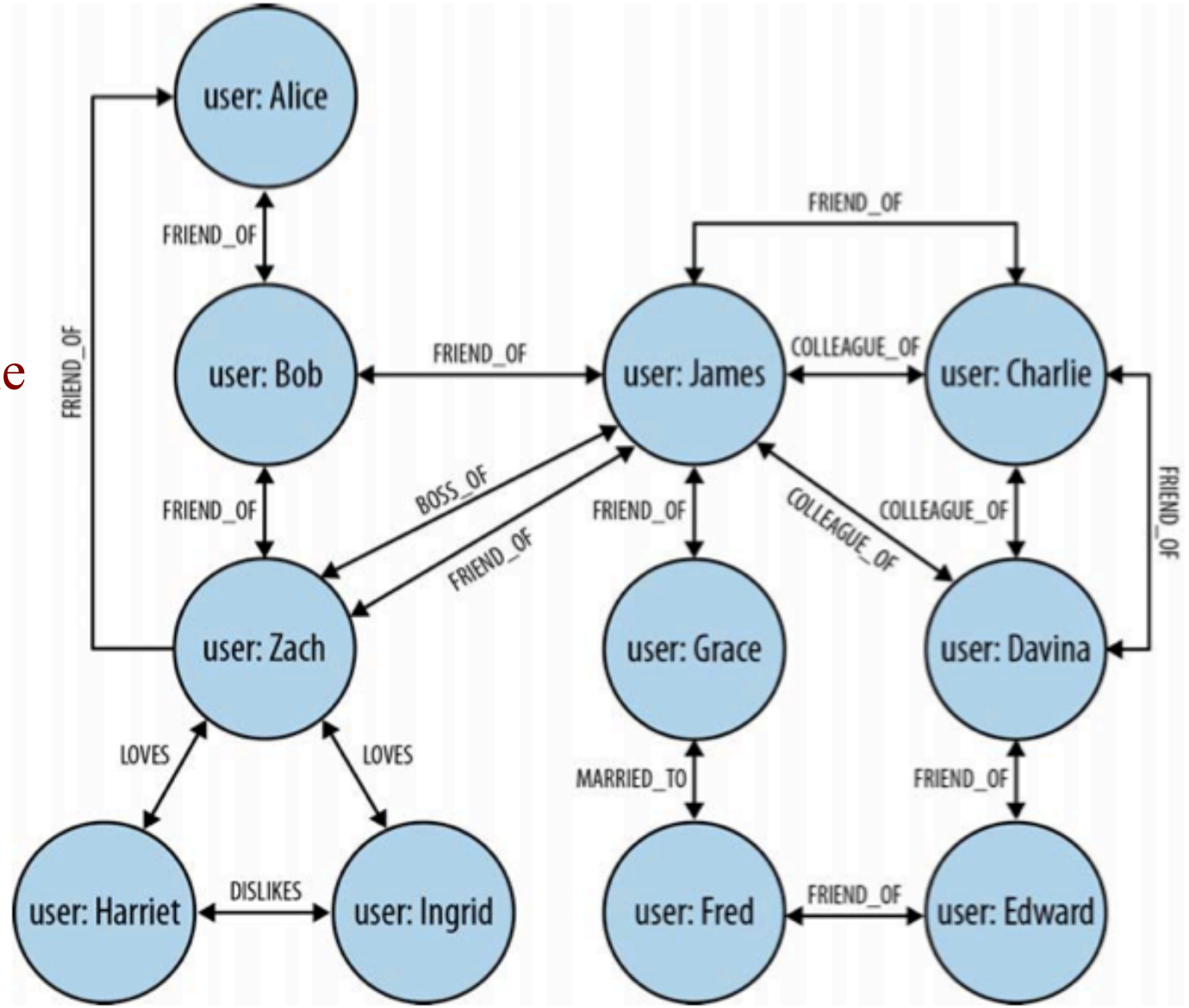
- Systems (e.g., Neo4j) are generally optimized for *transactional performance*, and tend to guarantee ACID properties.

# 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 dynamic information is natural and simple

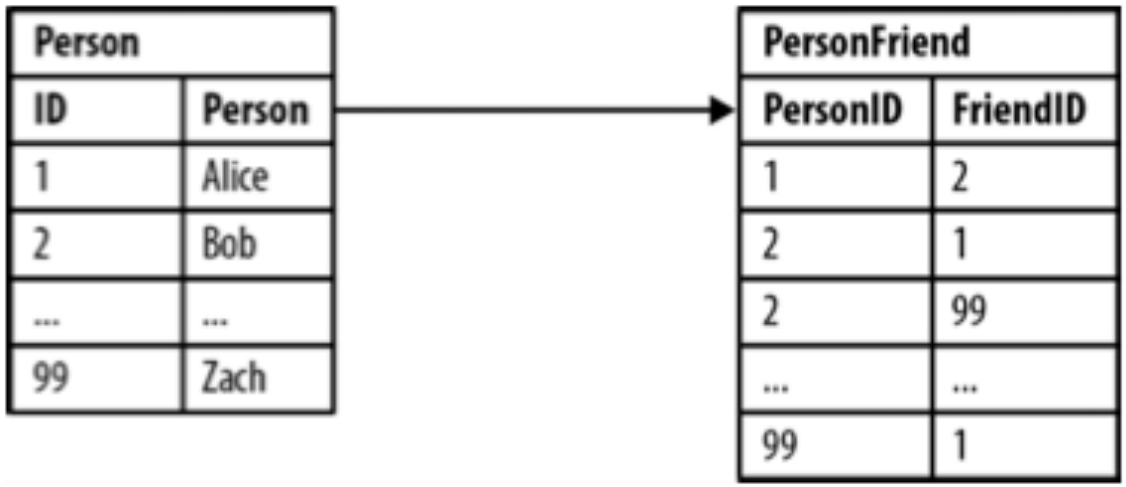


# Graph Databases Embrace Relationships

- 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 and data center management, 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 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



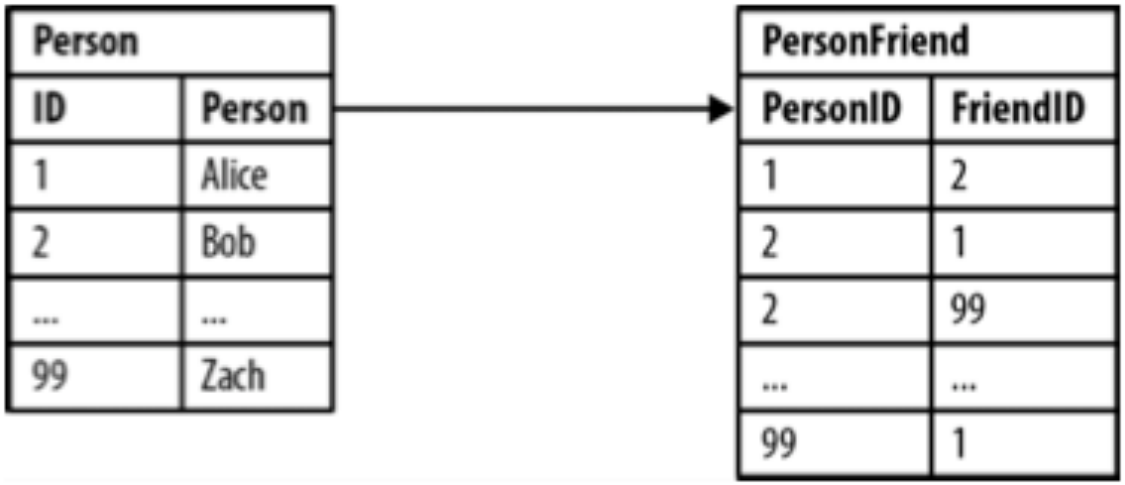
| Person | |
|---|---|
| ID | Person |
| 1 | Alice |
| 2 | Bob |
| ... | ... |
| 99 | Zach |

| PersonFriend | |
|---|---|
| PersonID | FriendID |
| 1 | 2 |
| 2 | 1 |
| 2 | 99 |
| ... | ... |
| 99 | 1 |

Notice that in this example, PersonFriend has not to be considered simmetric: Bob may consider Zach as friend, but the converse does not necessarily holds.

# Graph DBs vs Relational DBs- Example

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



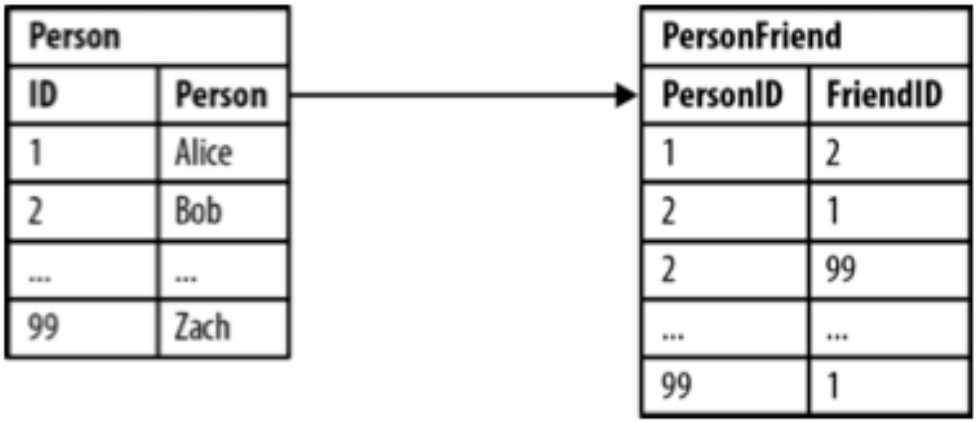
| Person | |
|---|---|
| ID | Person |
| 1 | Alice |
| 2 | Bob |
| ... | ... |
| 99 | Zach |

| PersonFriend | |
|---|---|
| PersonID | FriendID |
| 1 | 2 |
| 2 | 1 |
| 2 | 99 |
| ... | ... |
| 99 | 1 |

```
SELECT p2.Person
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, "who are *the 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 |

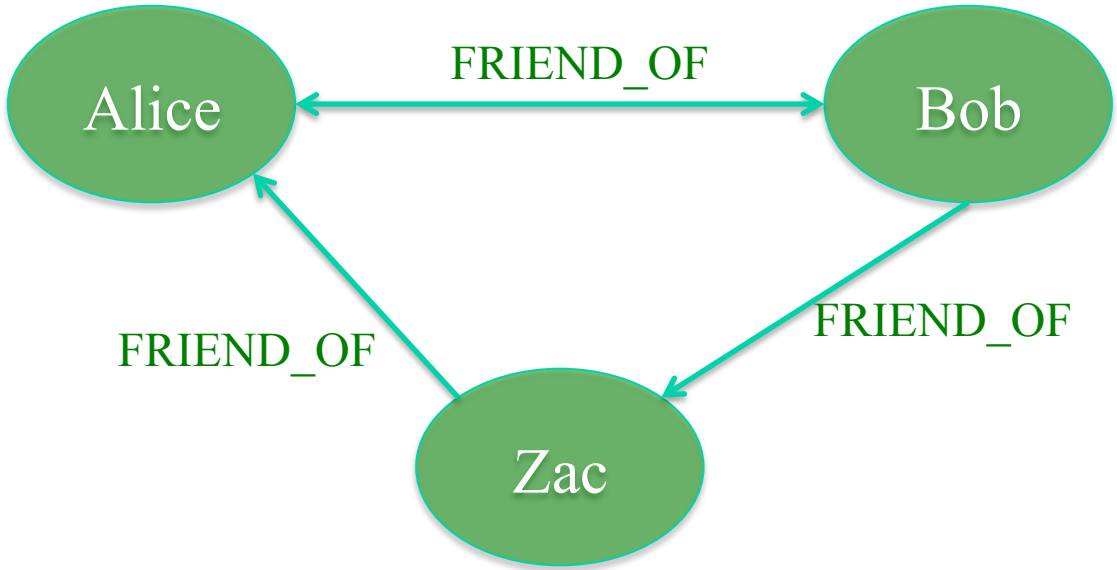
```
SELECT p2.Person AS FRIEND_OF_FRIEND
FROM Person p1 JOIN PersonFriend pf1 ON
     p1.ID = pf1.PersonID JOIN PersonFriend pf2 ON
     pf1.FriendID = pf2.PersonID JOIN Person p2 ON
     pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

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 (but other operations may be however more difficult).

# 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*

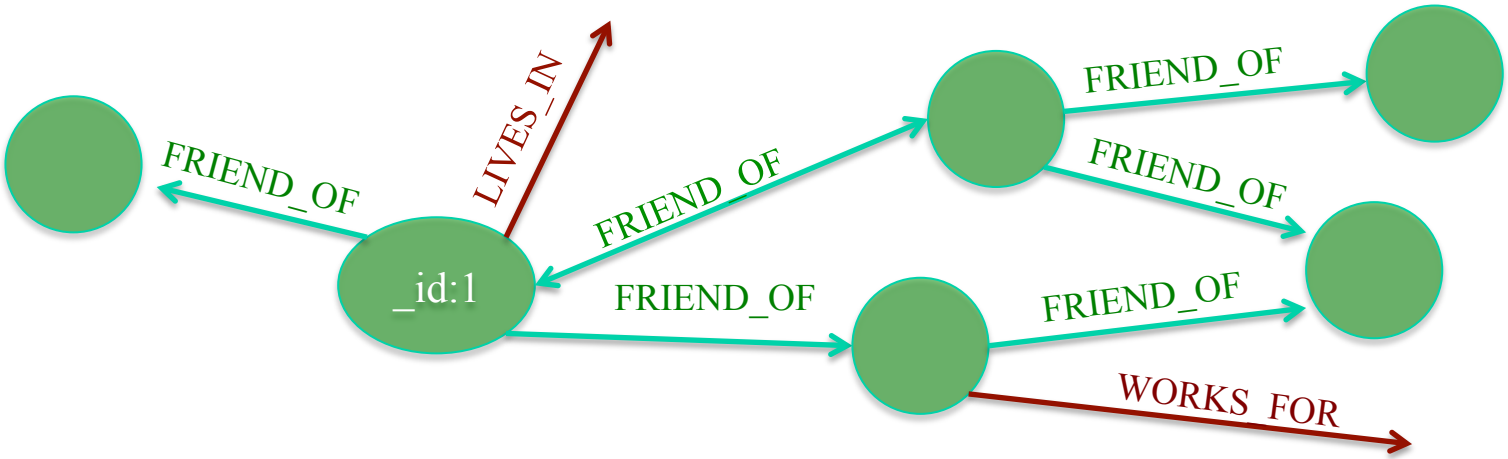
# Graph DBs vs Relational DBs - Costs

Consider again the Alice's friend-of-friend query (simplified, since we use only Ids and do not exclude the Alice's Id from the result)

```
SELECT pf2.FriendID AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN PersonFriend pf2 ON
     pf2.PersonID = pf1.FriendID
WHERE pf1.PersonID = 1
```

- If $m$ is the number of pairs of friends, this means a cost of $O(m^2)$
- Indexes reduce this cost, since they allow us to avoid linear search over a column.
- However, if the number of joins increase, the cost may become unaffordable (and depends on the size of the entire tables involved).

# Graph DBs vs Relational DBs - Costs




- 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 searching for all the reached nodes.
- If $p$ bounds the number of outgoing edges ($p<<n<m$, where $n$ is the number of persons and $m$ is the number of pairs of friends), and $k$ bounds the number of FRIEND_OF edges outgoing from a node ($k<<m$), then the cost is $O(p) + O(k*p)$ (notice that the cost for traversing an edge is constant).
- Local indexes are normally used to speed up local search

# 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. 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 friends-of-friends), 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

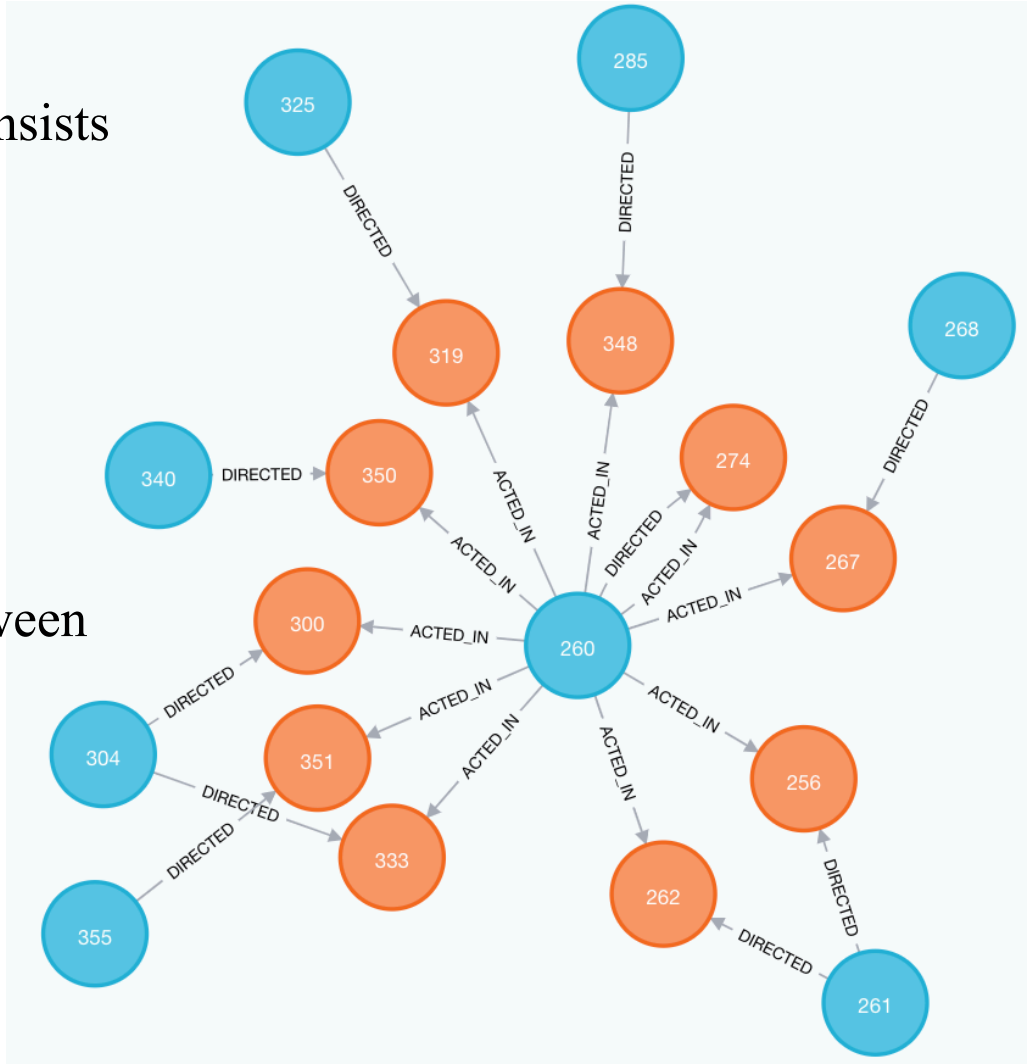
# Abstract Data Type

- G = (V,E) over a finite alphabet Σ consists
- 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 ⊆ V ×Σ×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)…}






Adapted From [ABFRV13]

# Basic Operations

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

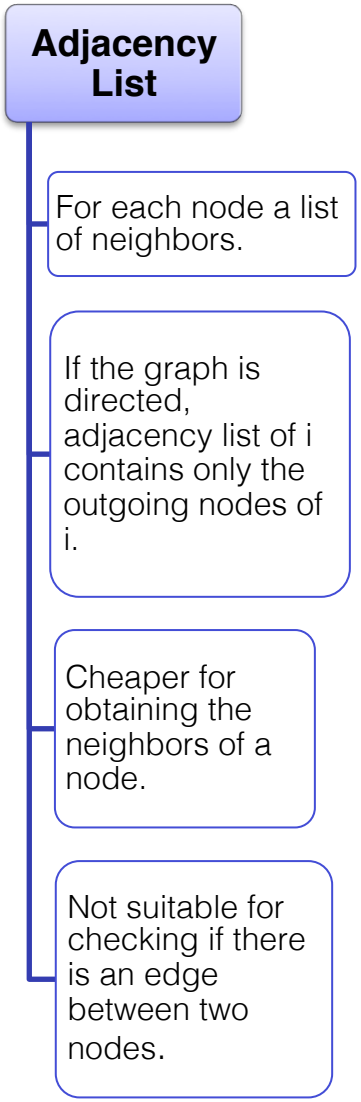
- AddNode(G,x): adds node x to the graph G.
- DeleteNode(G,x): deletes the node x from graph G.
- Adjacent(G,x,y): tests if there is an edge from x to y.
- Neighbors(G,x): returns nodes y s.t. there is an edge from x to y.
- AdjacentEdges(G,x,y): returns the set of labels of edges from x to y.
- Add(G,x,y,l): adds an edge between x and y with label l.
- Delete(G,x,y,l): deletes an 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]

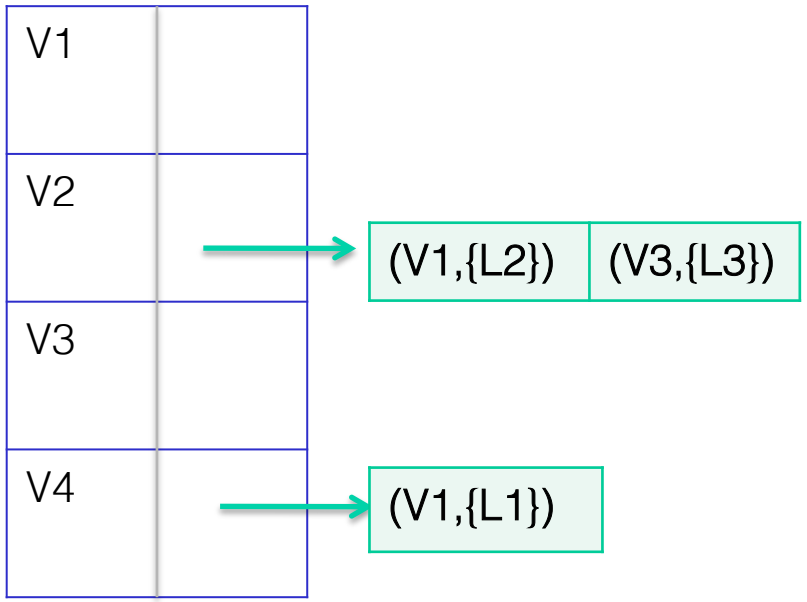
# Implementation of Graphs

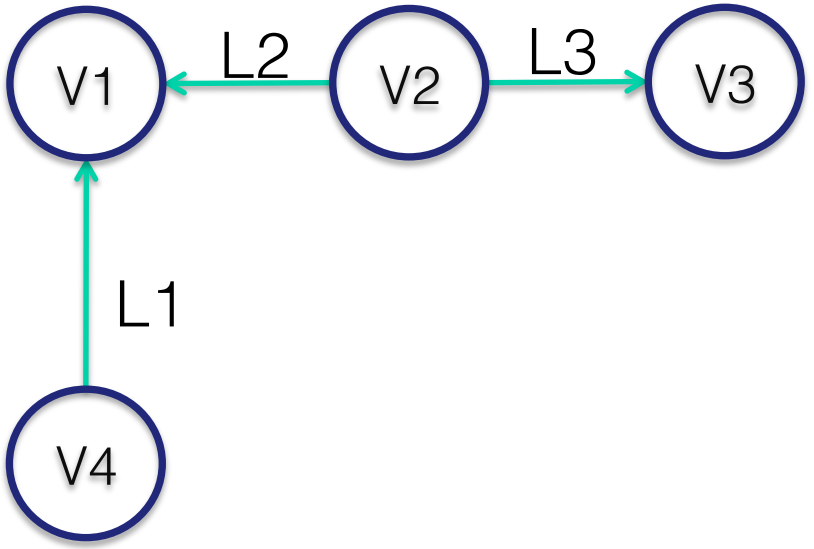





From [ABFRV13]

# Adjacency List




- Memory used is |V|+|E|
- 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**

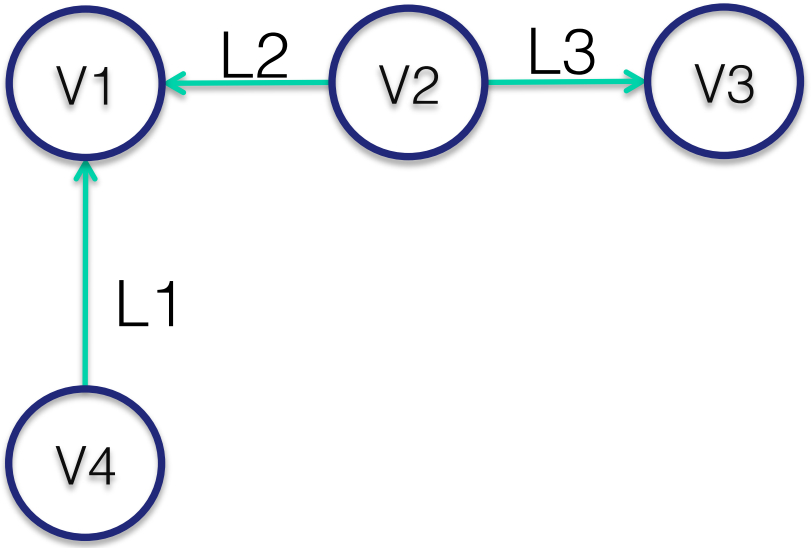
- Bidimensional graph representation.
- Rows represent source vertices.
- Columns represent destination vertices.
- Each non-null entry represents that there is an edge from the source node to the destination node.



From [ABFRV13]

# Adjacency Matrix

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



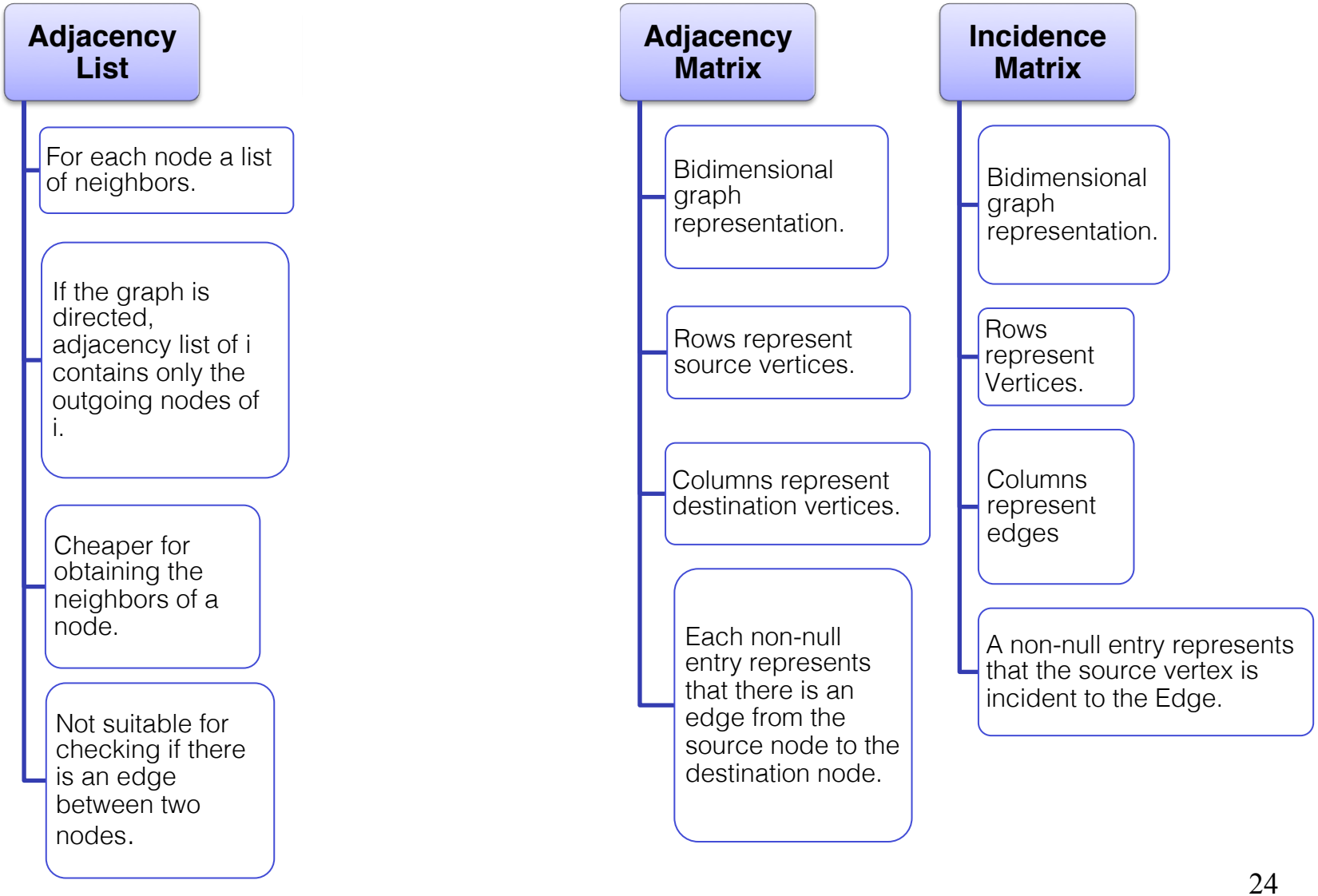

- Memory used is $|V|^2$
- Adding a vertex: add a row and column to the matrix. Removing a vertex: remove its row and column. The need of these operations makes the adjacency matrix unsuitable for graphs in which vertices are frequently added and removed. Adding and removing edges is easy however.
- 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 (it is a O(1) operation, whereas in Adjecency lists it is an O(|V|) operation).
- The matrix can be sparse





Adapted From [ABFRV13]

# Implementation of Graphs

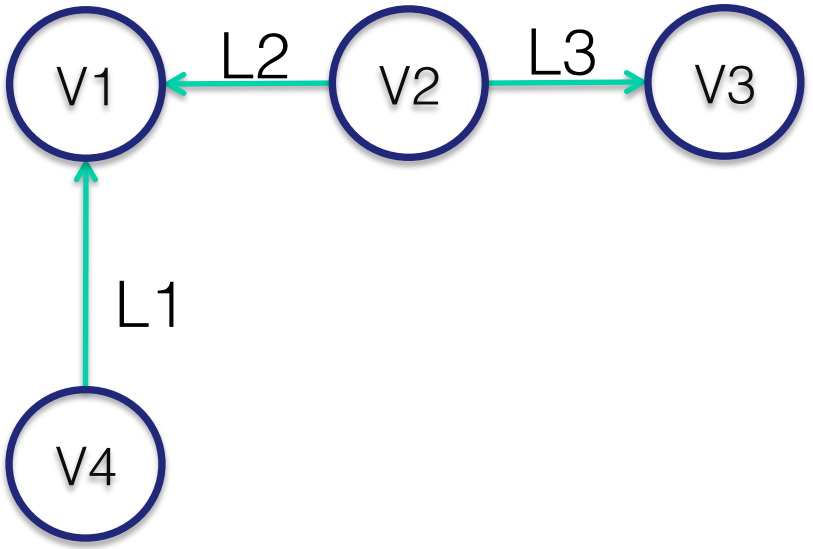





From [ABFRV13]

# Incidence Matrix*

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




- In this case the memory usage is vertices × labels: Storage: O(|V|x|E|)
- Adjacent(G,x,y): you have to scan the row of both the first and second vertex
- Neighbors(G,x): you have to scan the entire matrix
- AdjacentEdges(G,x,y): similar to Adjacent(G,x,y)
- Add(G,x,y,l): you have to add a column l put ‘source’ in position (x,l) and ‘destination’ in position (y,l)
- Delete(G,x,y,l): you have to delete the column l

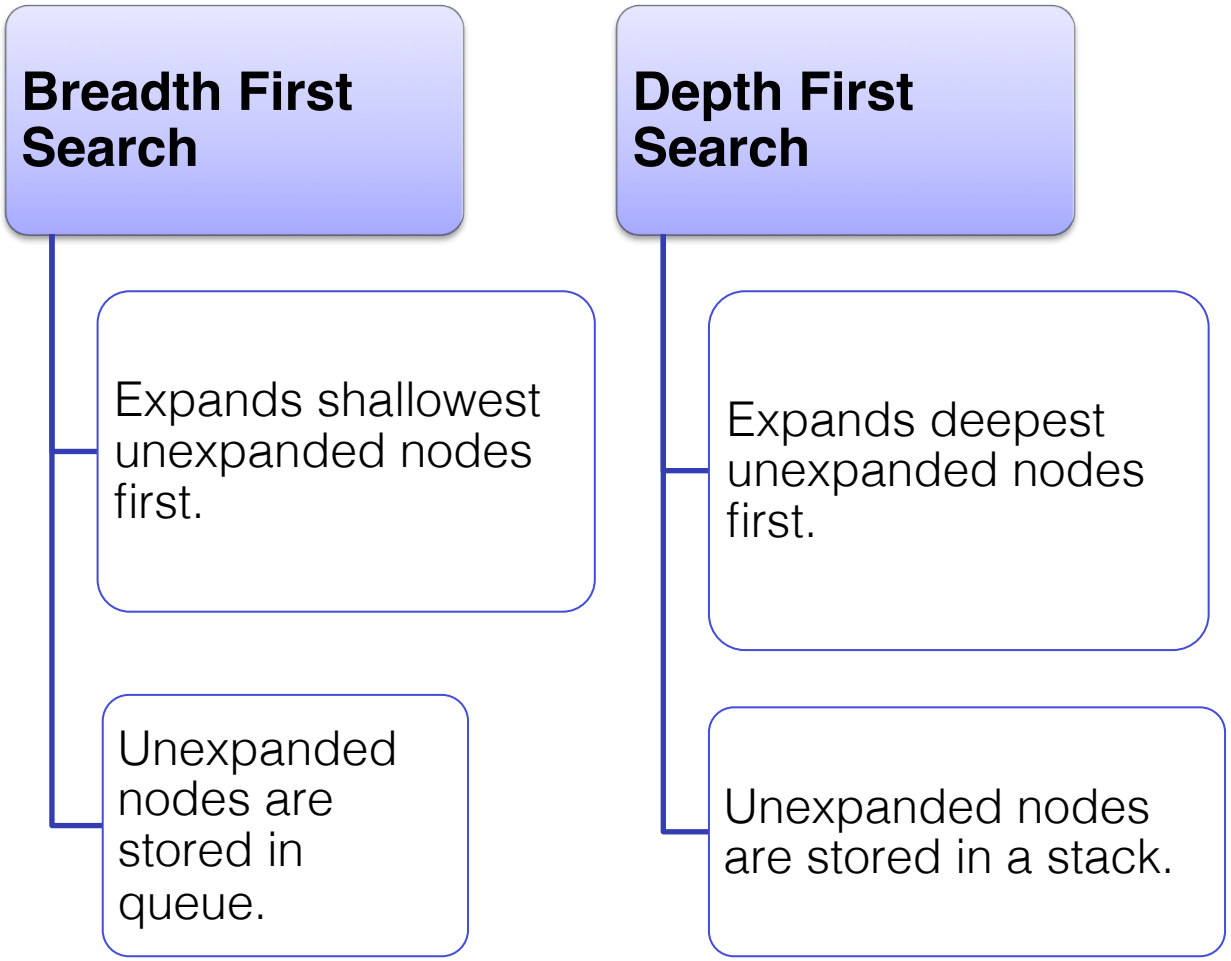
* *For simplicity in the example we use labels as identifiers of edges*





Adapted From [ABFRV13]

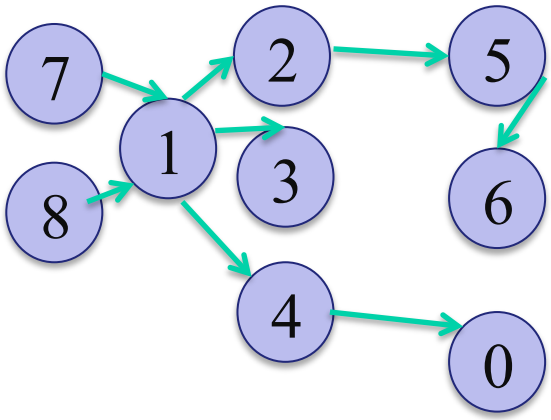
# Traversal Search

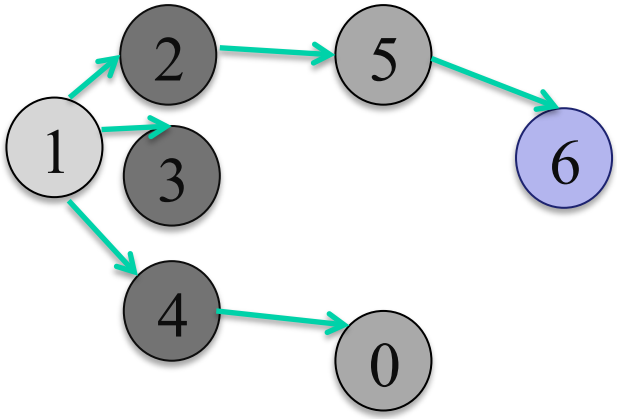





From [ABFRV13]

# Breadth First Search




**Notation:**

1 Starting Node

First Level Visited Nodes

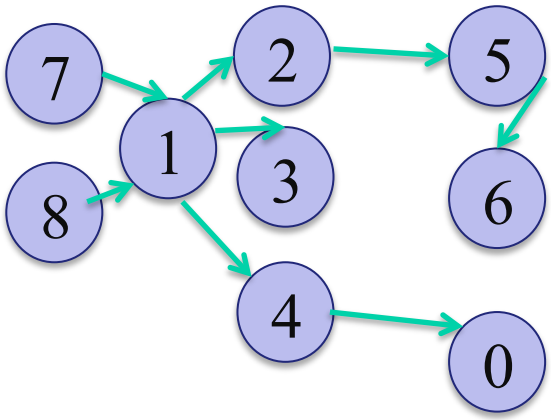
Second Level Visited Nodes

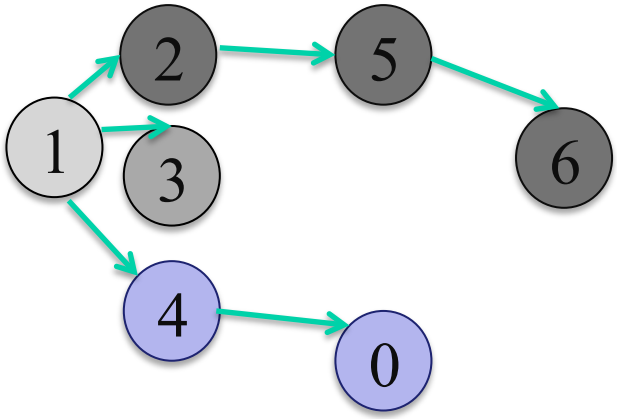
Third Level Visited Nodes



From [ABFRV13]

# Depth First Search




**Notation:**

1 Starting Node

First Level Visited Nodes

Second Level Visited Nodes

Third Level Visited Nodes

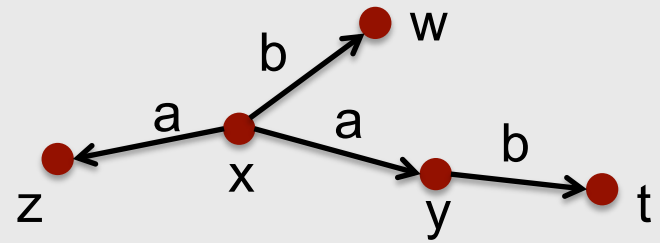


From [ABFRV13]

# 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 $\Sigma$ 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 $(v_0, a_1, v_1)$ $(v_1, a_2, v_2) \ldots (v_{m-1}, a_m, v_m)$, where $(v_{i-1}, a_i, v_i)$ is an edge in E, for each $1 \leq i \leq m$. The *label* of $\pi$, denoted $\lambda(\pi)$, is the string $a_1a_2...a_m \in \Sigma^*$.
- *A Regular path query* is a regular expression L over $\Sigma$, whose 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.

# Example




Graph G (Σ={a,b})

regular expression `L = ab*`

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

*Query languages for graph databases normally extend this class of queries*

# Regular Expressions

Syntax of regular expressions:

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

where

- `s` is an element of the alphabet $\Sigma$
- `·` 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:

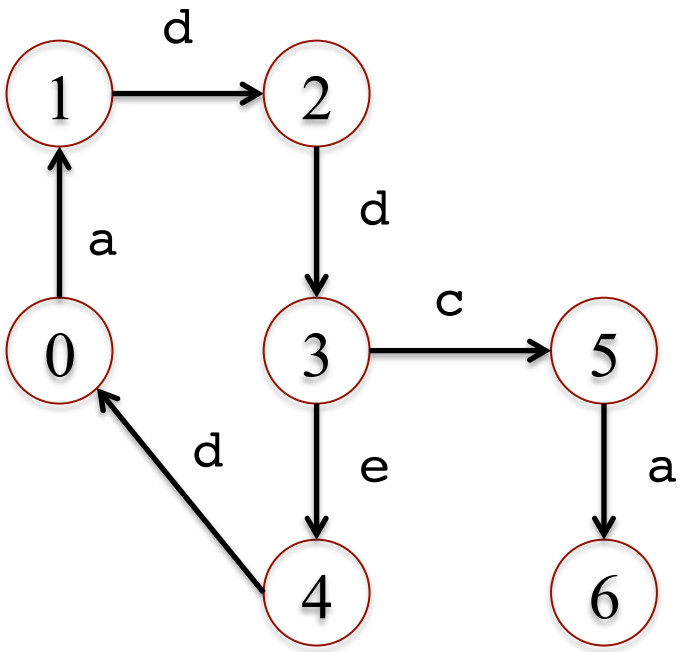
- Has ancestors:

  `isChildOf+`

- Are cousins (for simplicity, an individual can be cousin of hersef):

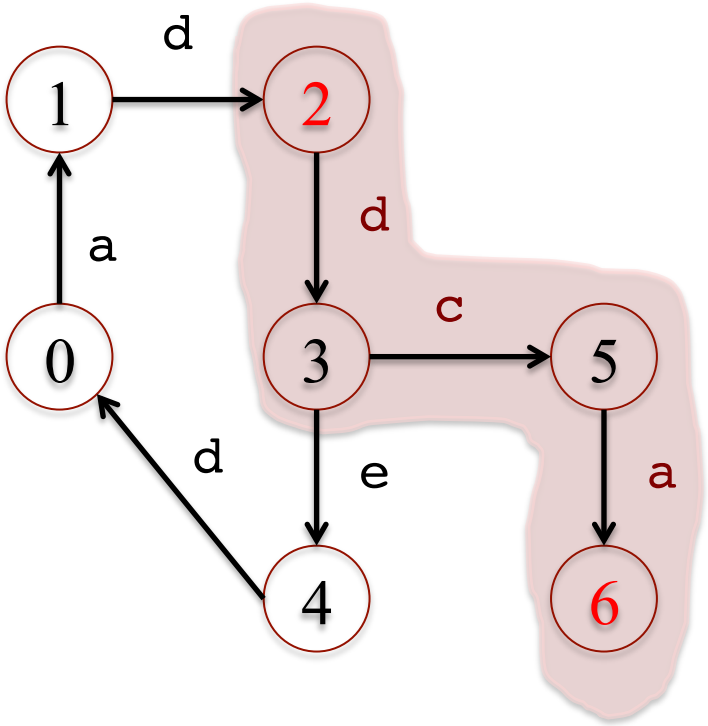
  `isChildOf·isChildOf·hasChild·hasChild`

# Example




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

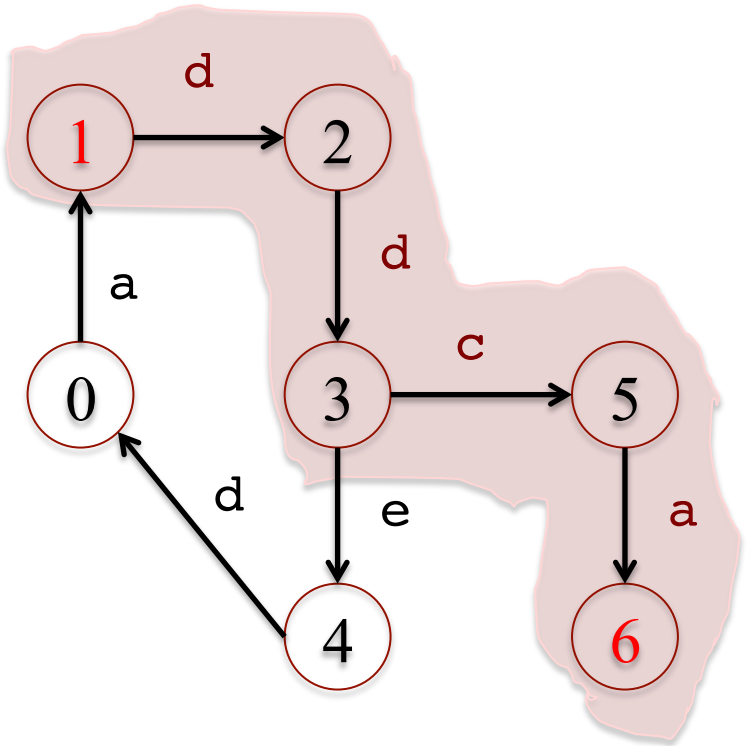
# Example




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

matching path: `dca`

# Example




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

matching path: `ddca`

# 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

# 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 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.

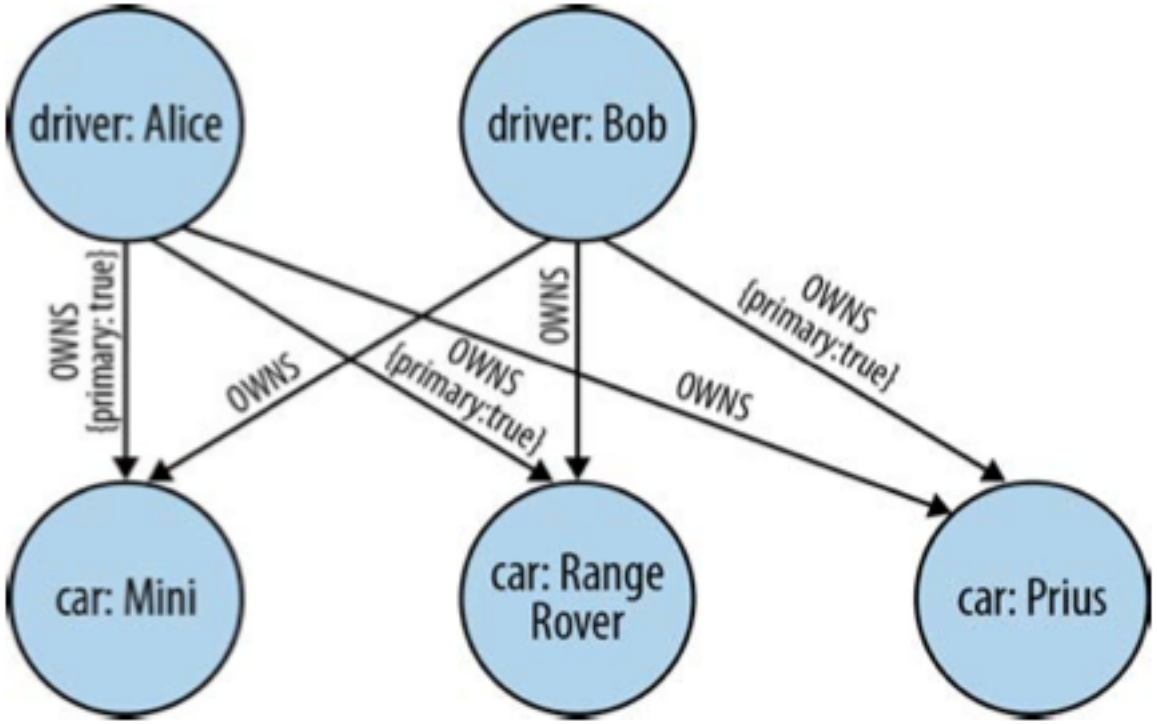
**Note:** Some authors consider index-free-adjacency a distinguishing property for graph databases (i.e., a DBMS not using index-free-adjacency is not a Graph DBMS). Alternatively (as we do in these slides) it is possible to classify as graph database any database that from the user's perspective *behaves*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

# Types of graph databases

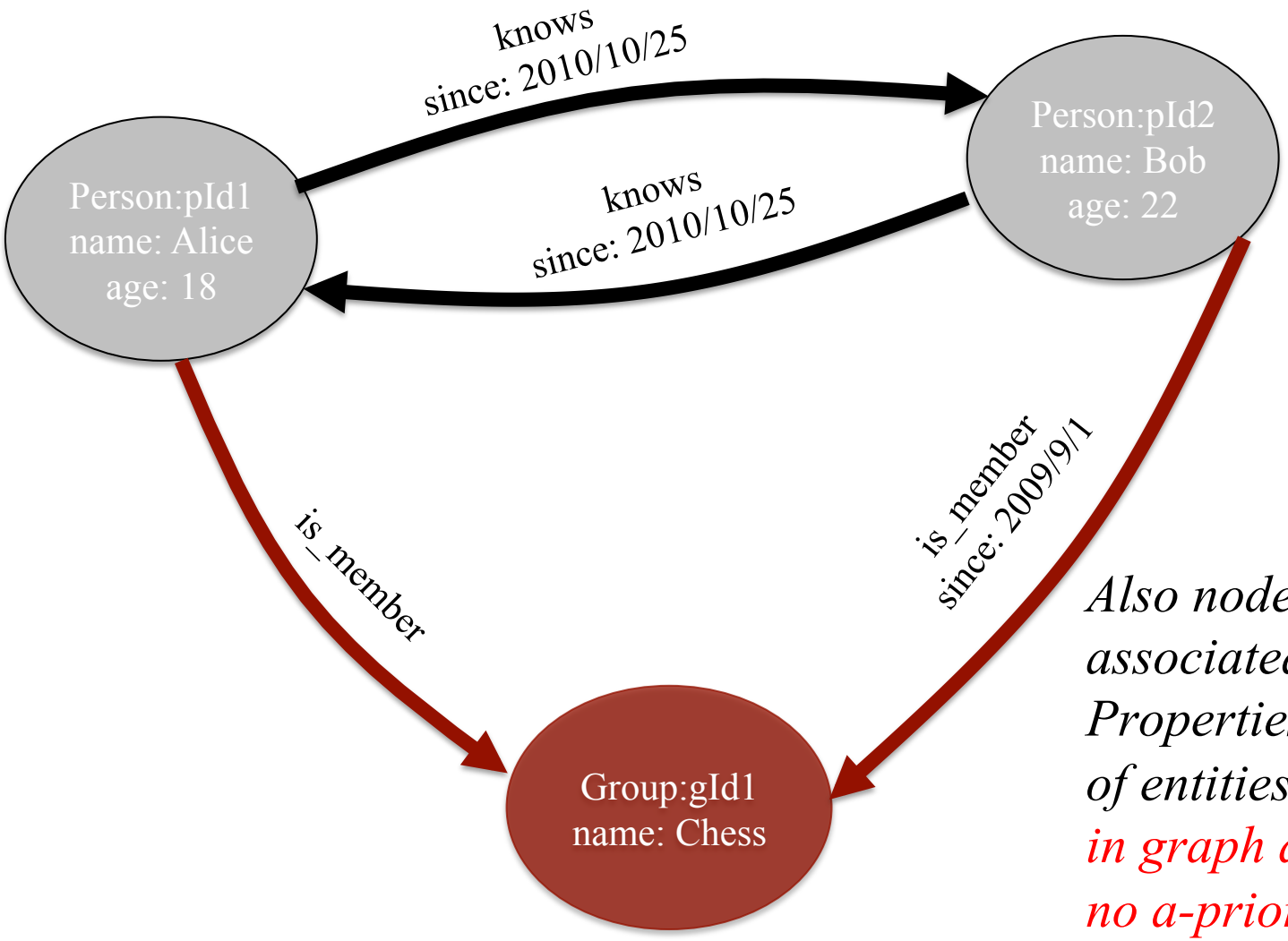
- There are several different graph data models, which somehow generalizes the basic definition we have seen before, including
  - property graphs,
  - hypergraphs,
  - triple stores.

# Property-graph databases

- A property graph is a labeled directed multigraph $G = (V, E)$ over an alphabet $\Sigma$ 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

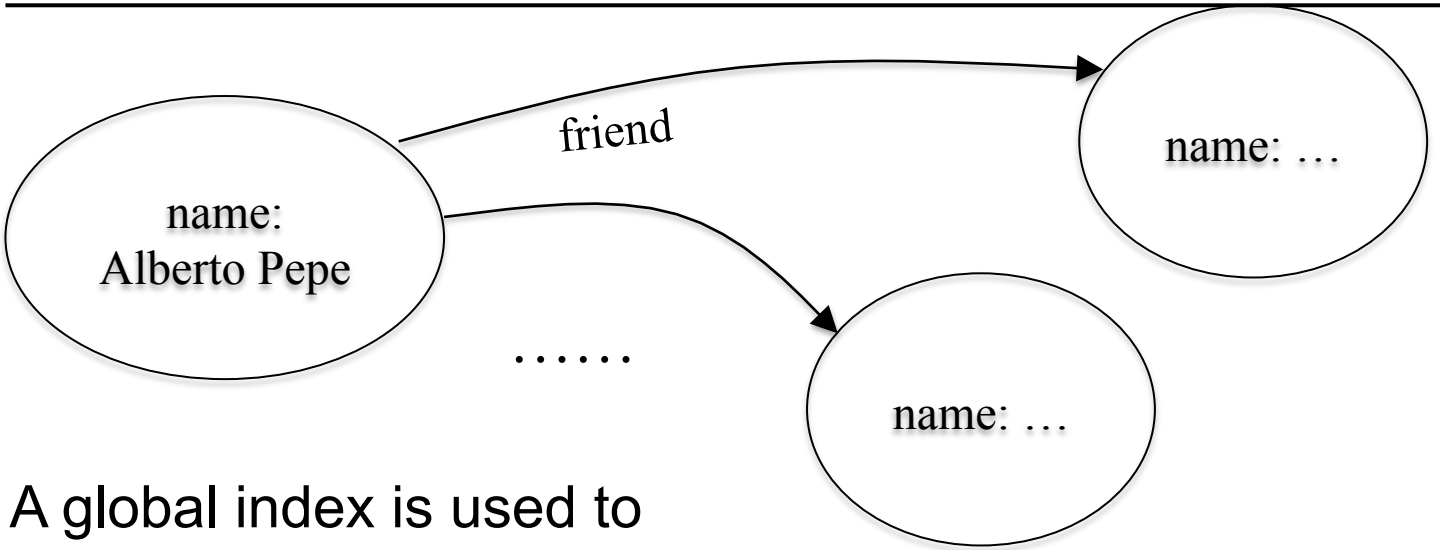



*Also nodes can be associated to properties. Properties act as attributes of entities or relatiships, but in graph databases there is no a-priori or rigid schema*

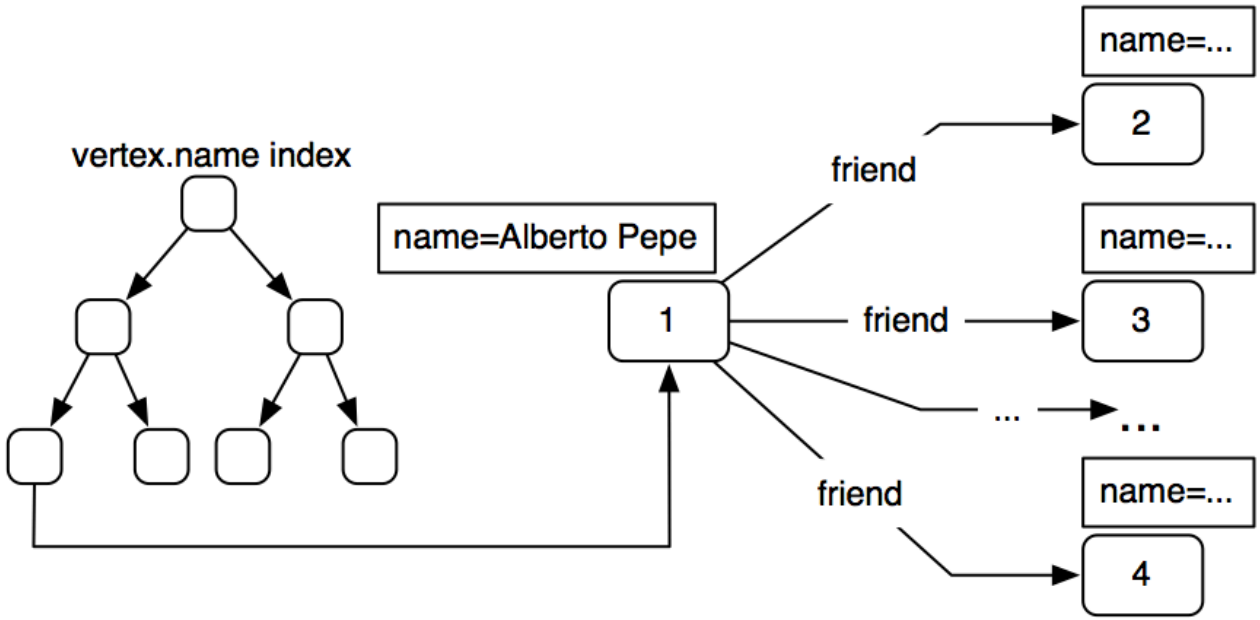
# Querying property-graph databases

- 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.

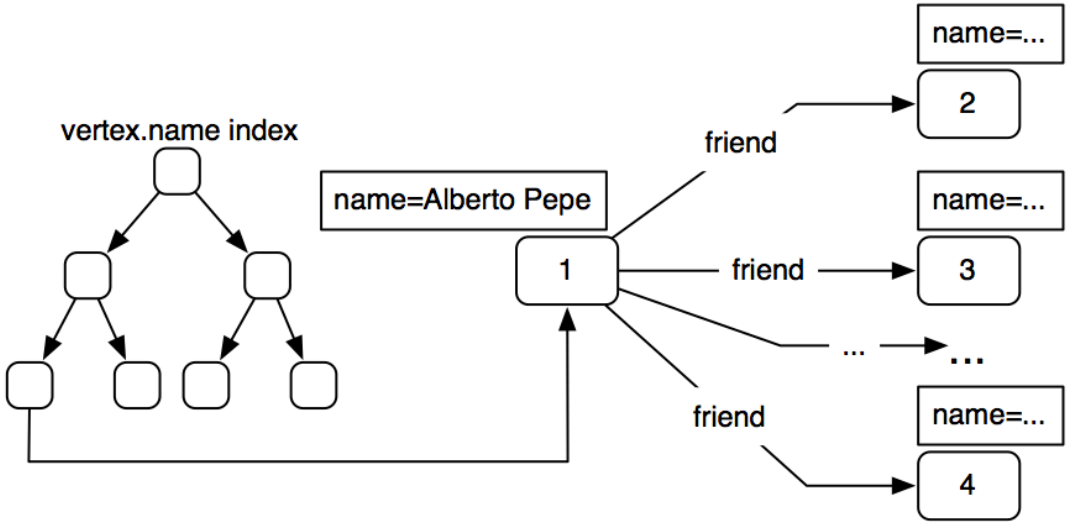
# People and their friends example




A global index is used to access vertices such that name= ‘Alberto Pepe’



# People and their friends example



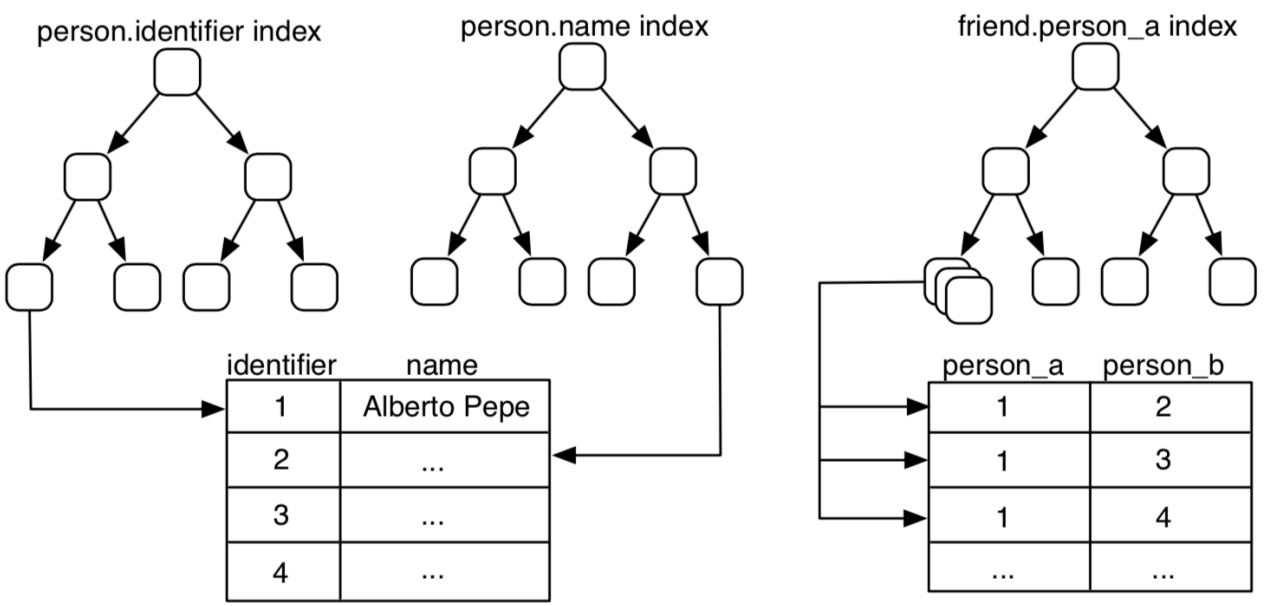

1. Query the vertex.name index to find all the vertices with the name "Alberto Pepe" [$O(\log_2 n)$] (where n is the number of nodes with the property name)
2. Given the vertex returned, get the k friend edges starting from this vertex. [$O(k + x)$] ] (where x is the number of the other outgoing edges)
3. Given the k friend edges retrieved, get the k vertices on the heads of those edges [$O(k)$] (getting one vertices is costant, 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 database

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

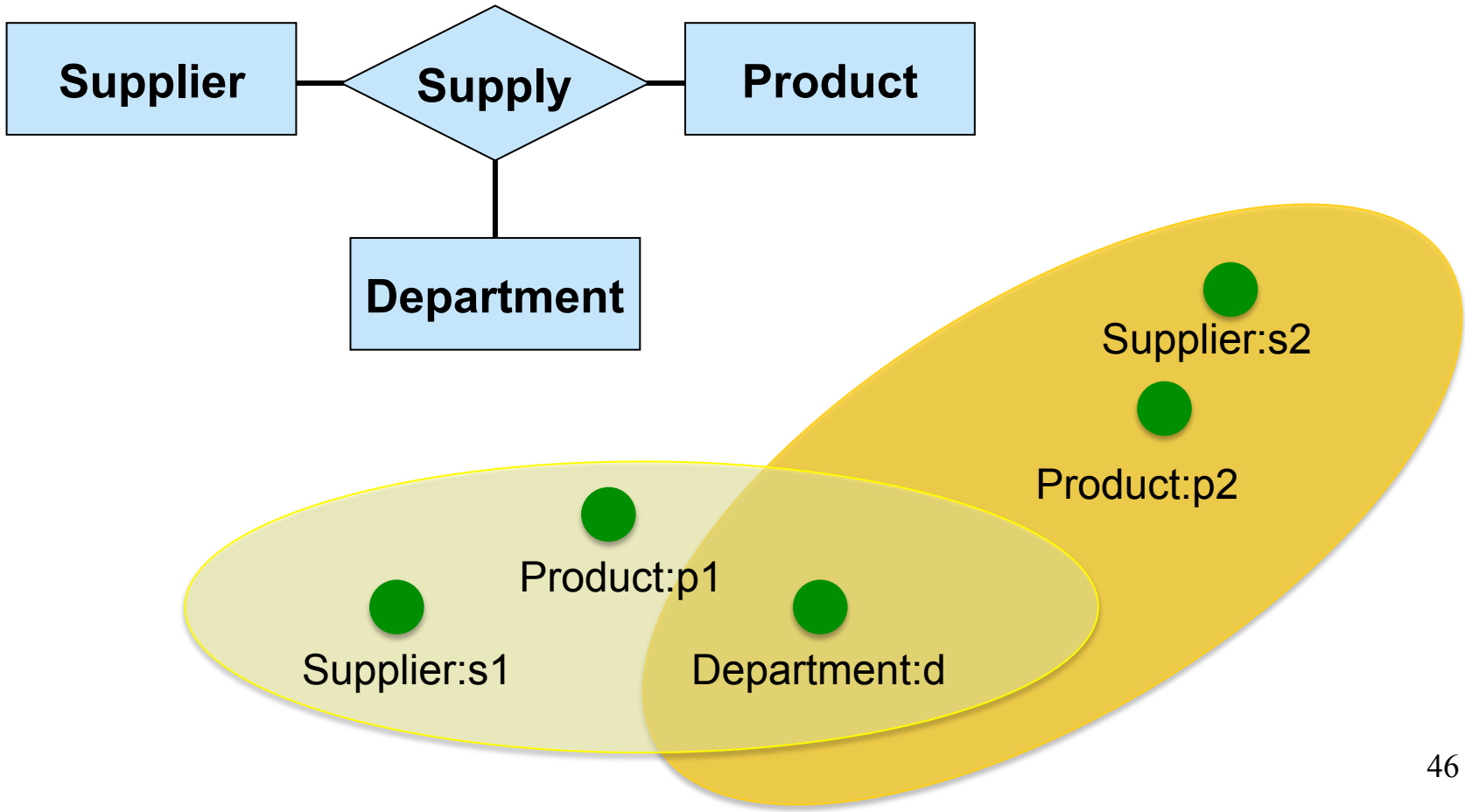
Suppose the problem of determining the name of all of Alberto Pepe's friends.

1. Query the person.name index to find the row in person with the name "Alberto Pepe" [$O(\log_2 n)$] (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 previous step [$O(\log_2 z)$] (z is the number of index nodes in friend.person_a)
4. Given each of the k rows returned, get the person_b identifier for those rows [$O(k)$]
5. For each k friend identifiers, query the person.identifier index for the row with friend identifier. [$O(k \log_2 n)$]
6. Given the k person rows, get the name value for those rows. [$O(k)$]



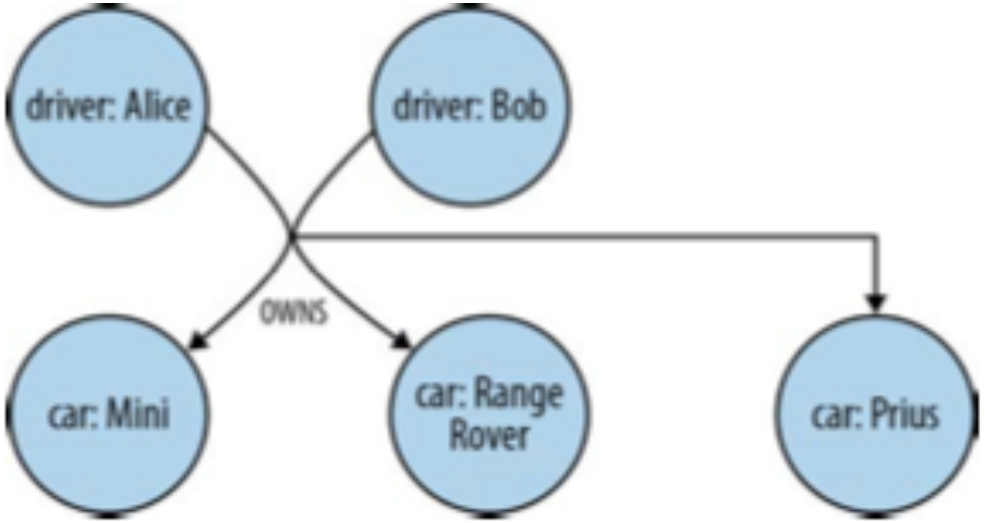
# Hyper-graph databases

- 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

# 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:** 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.

# Graph Databases

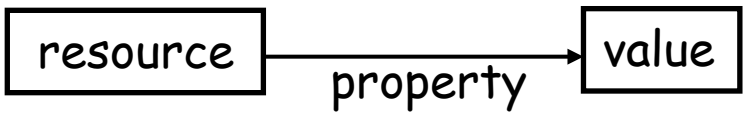
- Introduction to Graph Databases
- Resource Description Framework (RDF)
- RDF storage
- Querying RDF databases: The SPARQL language
- Linked data
- Tools

# Resource Description Framework

- 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 model = set of RDF triples
- triple = expression (statement)

(subject, predicate, object)

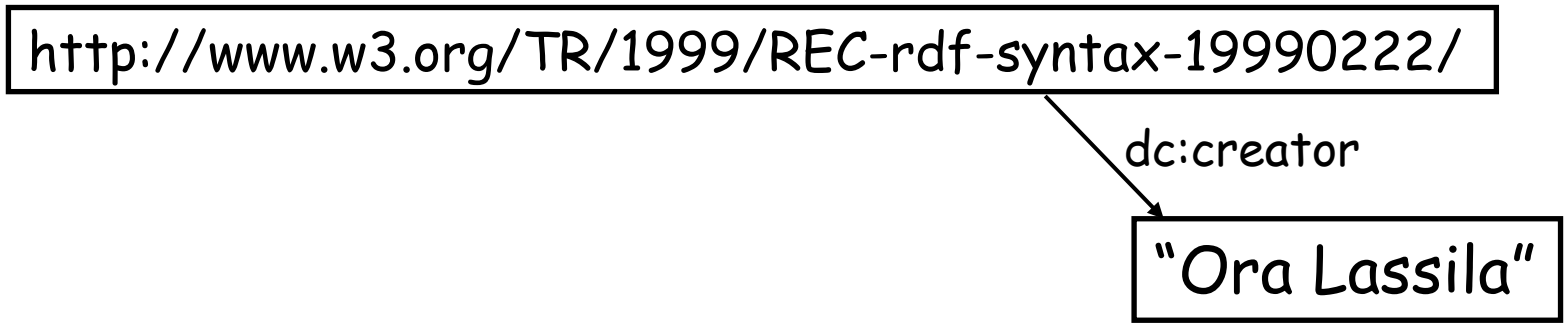
- subject = resource
- predicate = property (of the resource)
- object = value (of the property)



# RDF triples

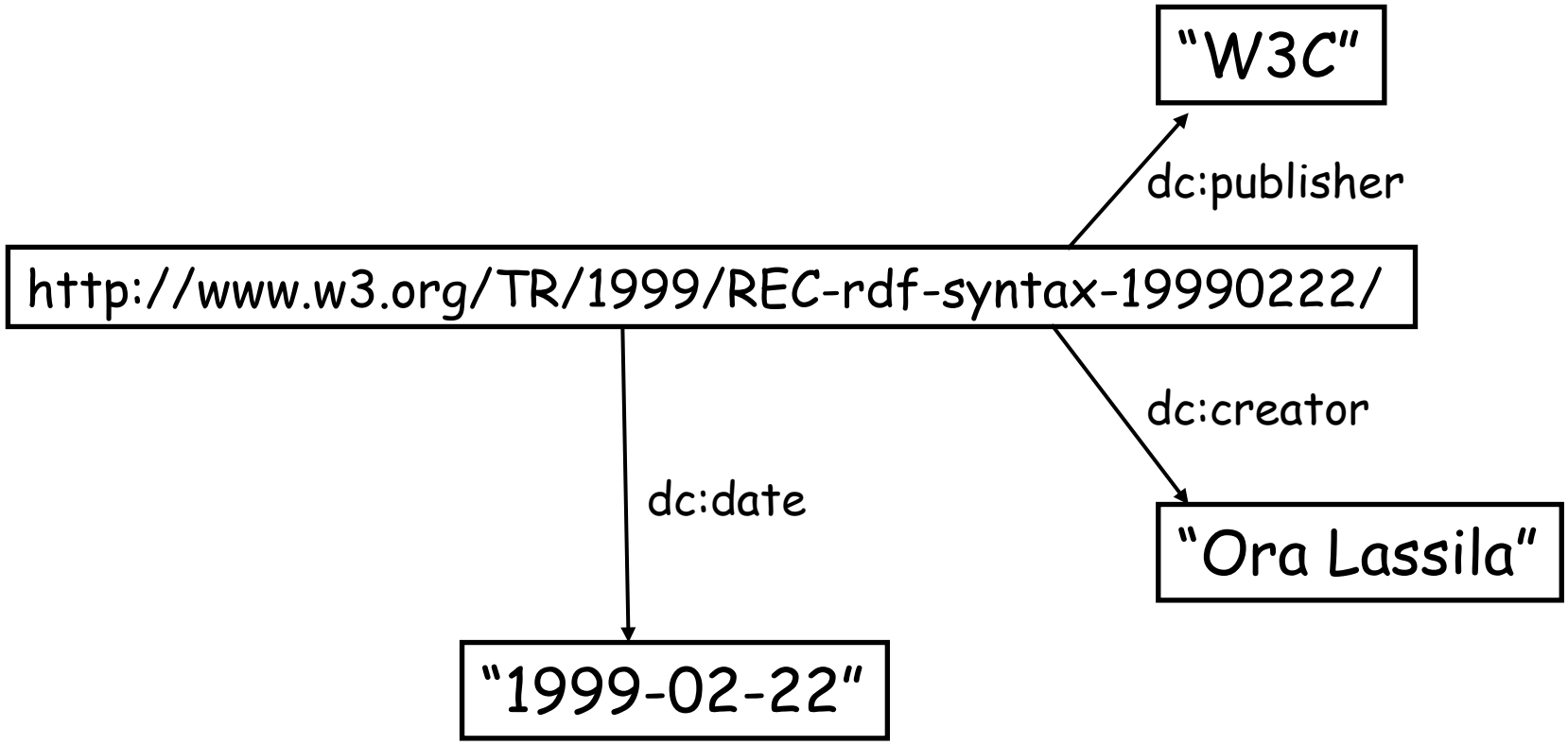
example: "the document at
`http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`
has Ora Lassila as creator"

triple:
`http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ creator "OraLassila"`




⇒ 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
- 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.

# 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:titlte "graph databases"^^xsd:string) - explicit type string

Other types:

- xsd:decimal
- xsd:integer
- 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

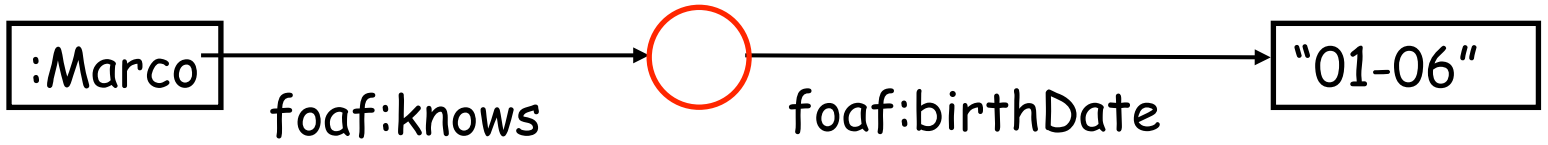
- Dublin Core (`dc:` or `dcterms:`) – `http://purl.org/dc/terms/`
- FOAF (`foaf:`) – `http://xmlns.com/foaf/0.1/`
- RDFS (`rdfs:`) – `https://www.w3.org/2000/01/rdf-schema`

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)

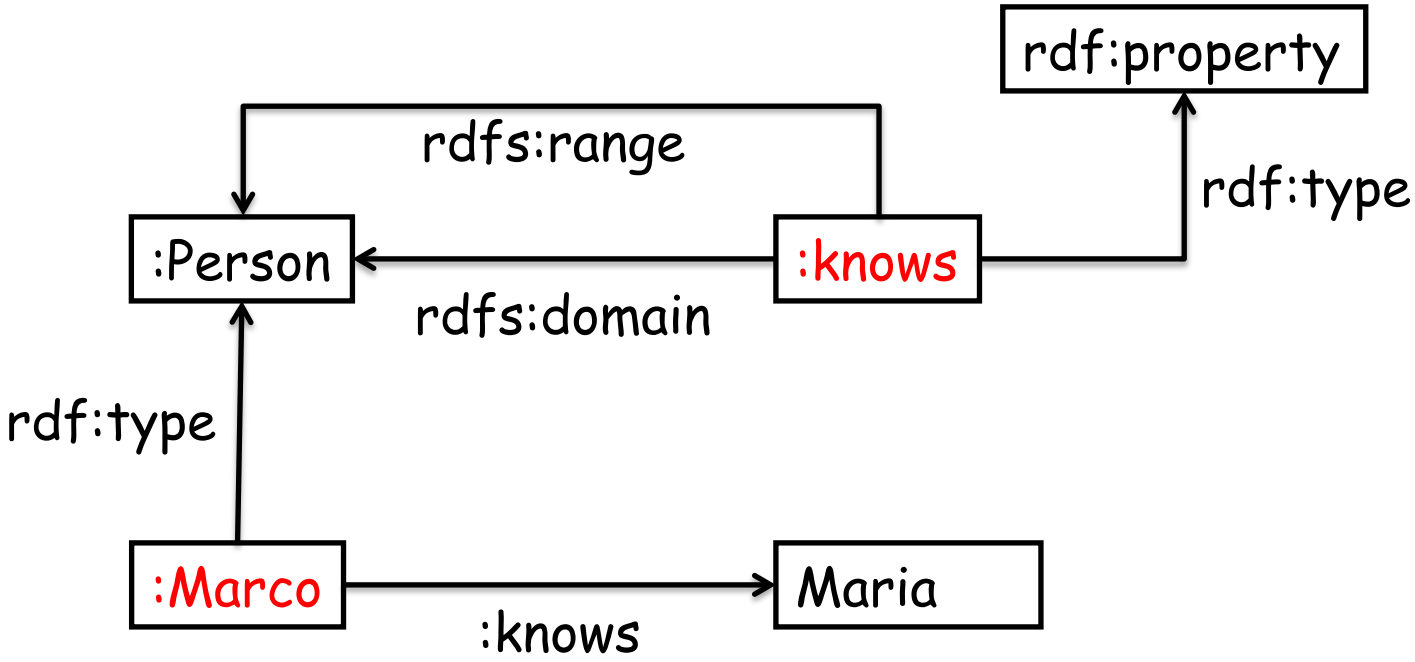
Example: Marco knows someone which was born on the Epiphany day




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

# 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

⇒ 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 represents 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:

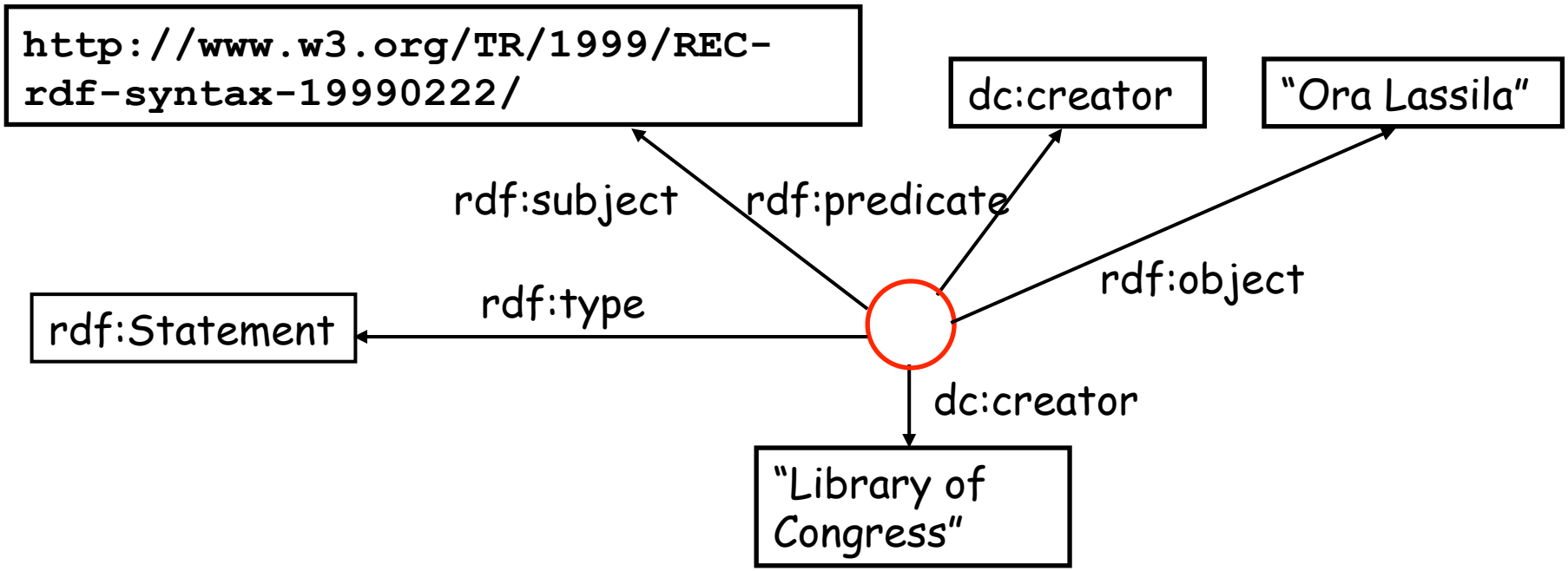
```
_:x rdf:type rdf:statement.
_:x rdf:predicate dc:creator.
_:x rdf:subject http://www.w3.org/TR/1999/REC-rdf-
                  syntax-19990222/.
_:x rdf:object "Ora Lassila".
```

- 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



# 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: a document is a simple set of triple of the form

```
subject predicate object.
```

- Turtle (Terse RDF Triple Language) notation.
  *Example (others in the next slides):*

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix : <http://myPrefix/>.
:mary rdf:type <http://www.ex.org/Gardener>.
:mary :worksFor :ElJardinHaus.
:mary :name "Dalileh Jones"@en.
_:john :worksFor :ElJardinHas.
_:john :idNumber "54321"^^xsd:integer.
```

  symbols preceded by _: denote blanks

# 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)

# 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 N3 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)

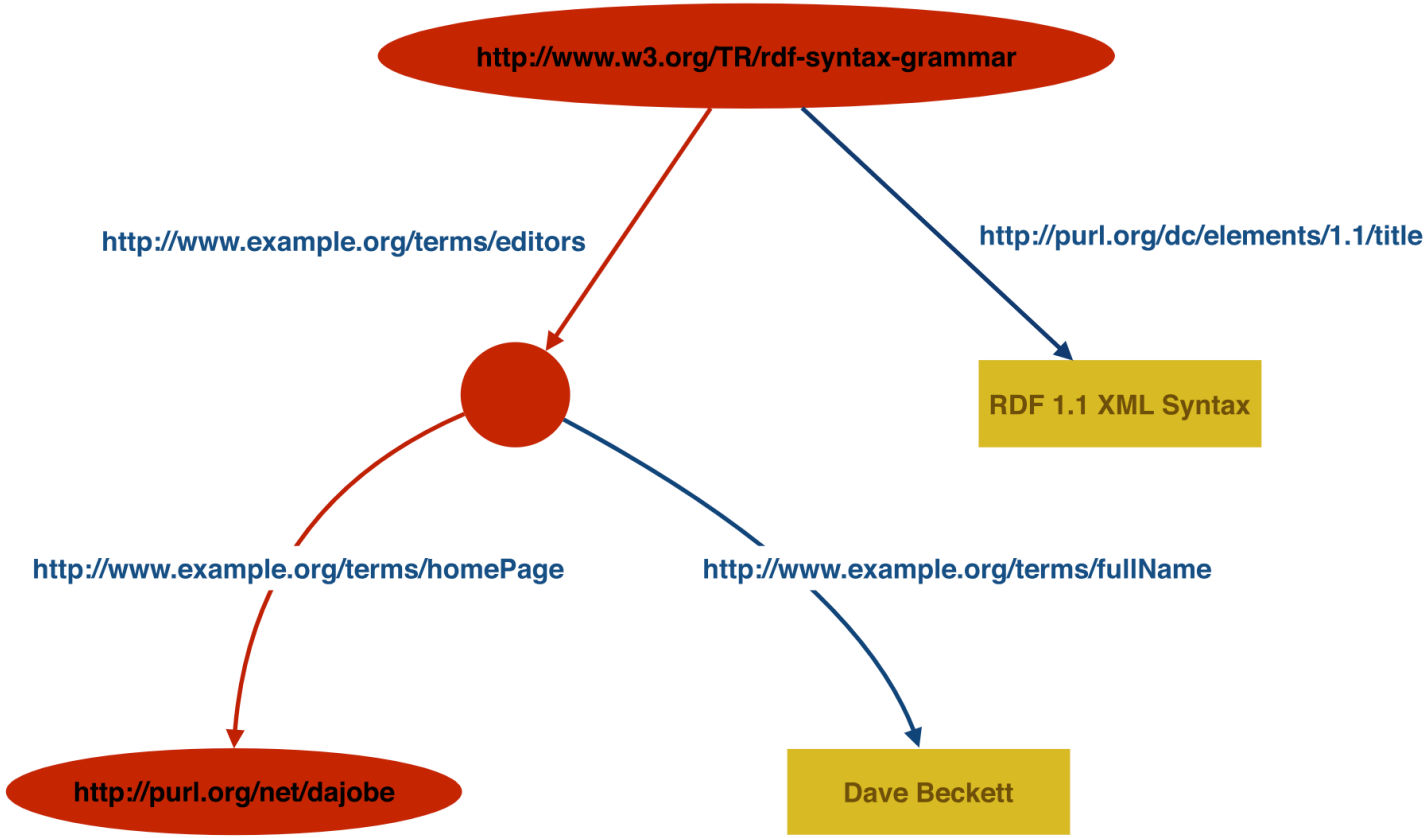
# RDF/XML syntax

- A node that represents a resource (labeled or not) is represented by an XML element whose 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 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 type is `rdf:Description`
- Values (literals) can be assigned with a 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/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:editors>
    <rdf:Description>
      <ex:homePage>
        <rdf:Description rdf:about="http://purl.org/net/dajobe/"></rdf:Description>
      </ex:homePage>
     <ex:fullName>Dave Beckett</ex:fullName>
    </rdf:Description>
  </ex:editors>
 ……
</rdf:Description>
```

# RDF Schema

RDFS = RDF Schema

- Defines a 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`*** (a resource is an instance of a class)
  - **`rdfs:subClassOf`** (a resource is a subclass of another resource)

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

# RDFS

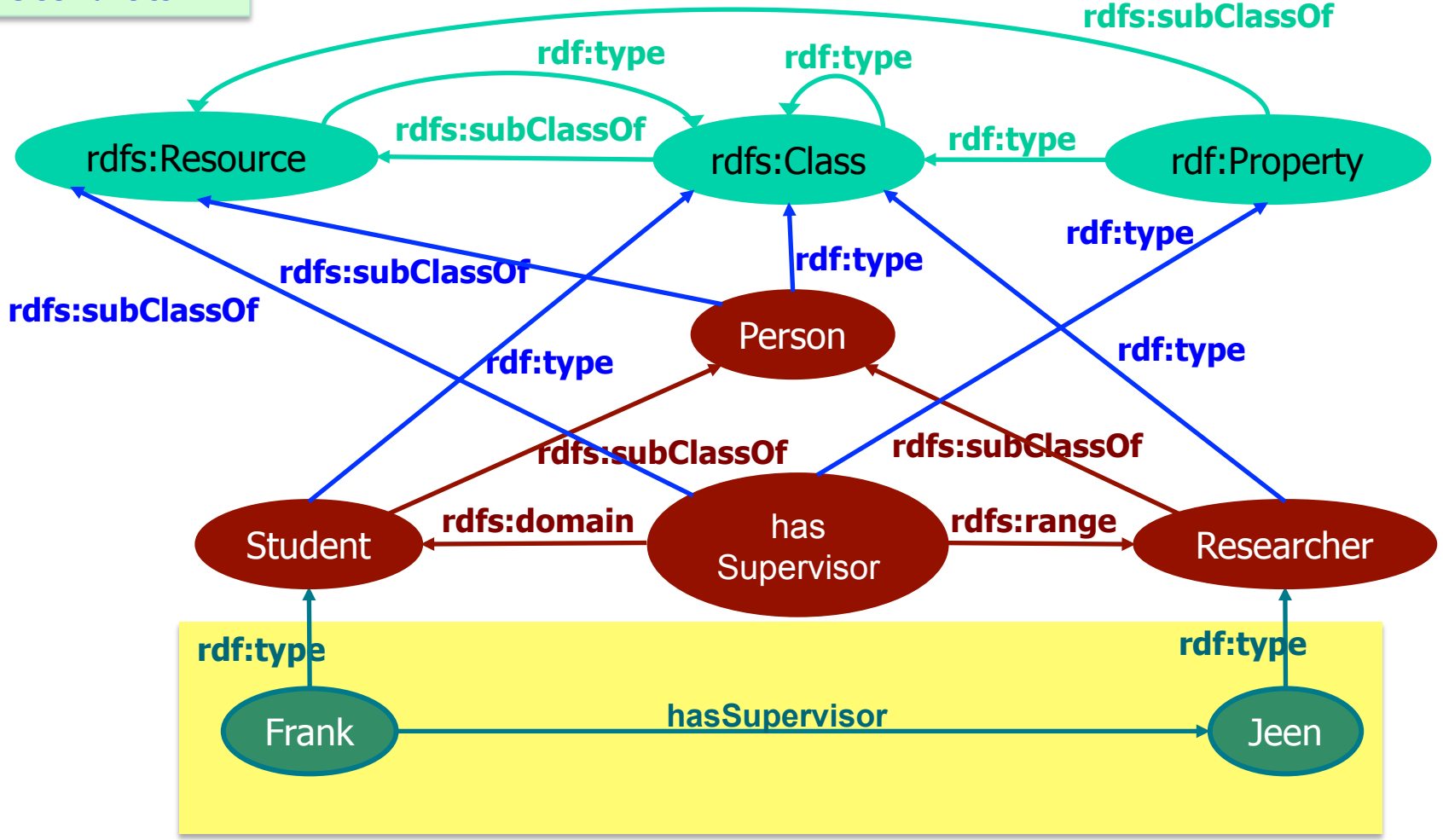
vocabulary for properties:

- **`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.
```

# 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
- Afterwards, a model-theoretic semantics has been provided

⇒formal definition of entailment and query answering over RDF(S) graphs

# 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

$\Rightarrow$ but no intuitive semantics

# Graph Databases

- Introduction to Graph Databases
- Resource Description Framework (RDF)
- RDF storage
- Querying RDF databases: The SPARQL language
- 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 thechniques 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 an 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*

{⟨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⟩, . . . }

*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 essentially 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 |

user3789

| | |
|---|---|
| name | Umi |
| birthdate | 1980 |
| likes | work1234 |
| $friendOf^{-1}$ | user8604 |

Debussy

| | |
|---|---|
| style | impressionist |
| $Composer^{-1}$ | work5678 |

Schoenberg

| | |
|---|---|
| style | expressionist |
| $Composer^{-1}$ | work1234 |

work1234

| | |
|---|---|
| Title | La Mer |
| MediaType | LP |
| Composer | Debussy |
| $likes^{-1}$ | user8604 |

user8604

| | |
|---|---|
| name | Teppei |
| birthdate | 1975 |
| phone | 2223334444 |
| phone | 5556667777 |
| likes | work5678 |
| likes | work1234 |
| friendOf | user3789 |

# 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.

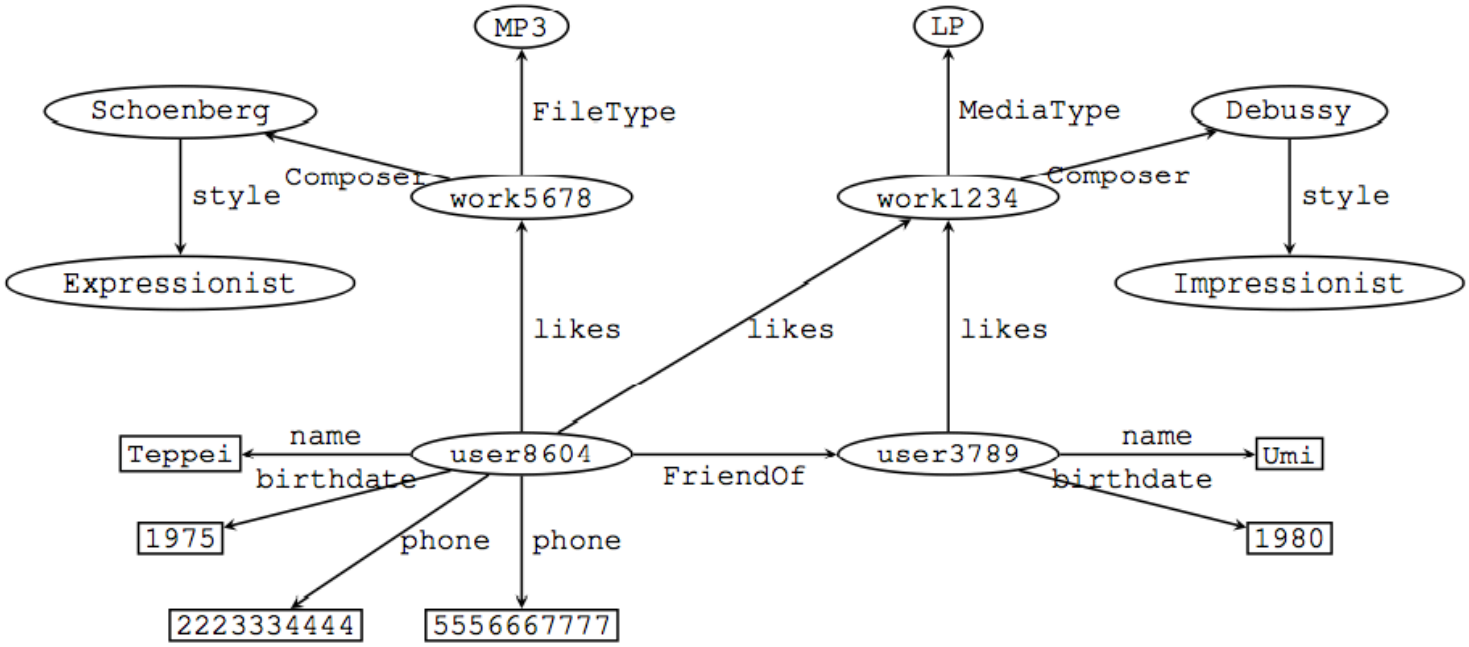
# The graph-based perspective for storing RDF graphs

- 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



# Graph Databases

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

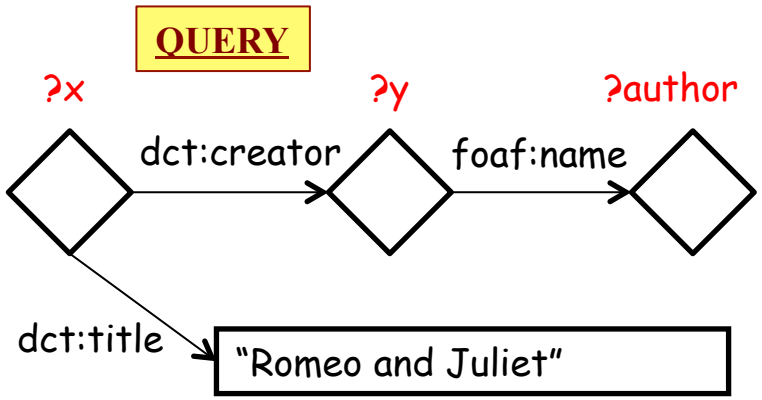
# SPARQL – example

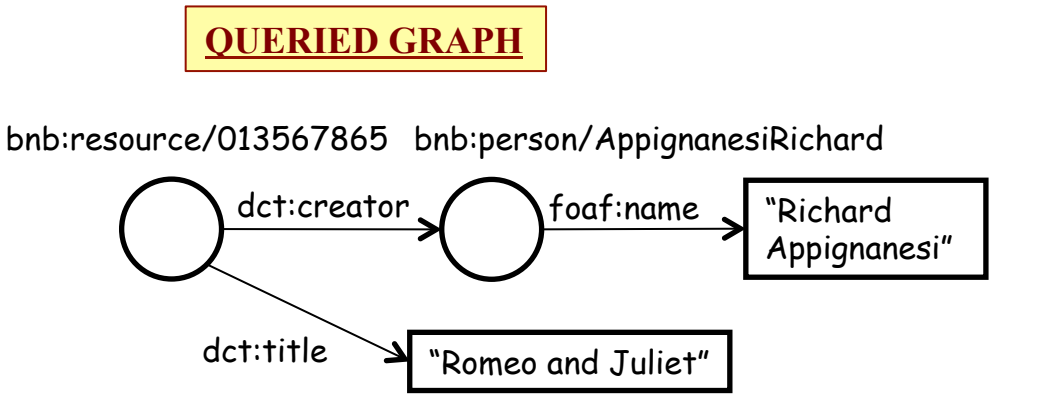
```
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?author
FROM <http://bnb.data.bl.uk/id/data/BNB>
WHERE { ?x dct:creator ?y.
        ?x dct:title "Romeo and Juliet".
        ?y foaf:name ?author}
```

- Variables are outlined through the "?" prefix ("$" is also possible).
- The `?author` variable will be returned as result.
- The `FROM` clause (optional) specifies the URI of the graph to be queried
- The SPARQL query processor returns all hits matching the pattern of the four RDF-triples.

# SPARQL – query evaluation

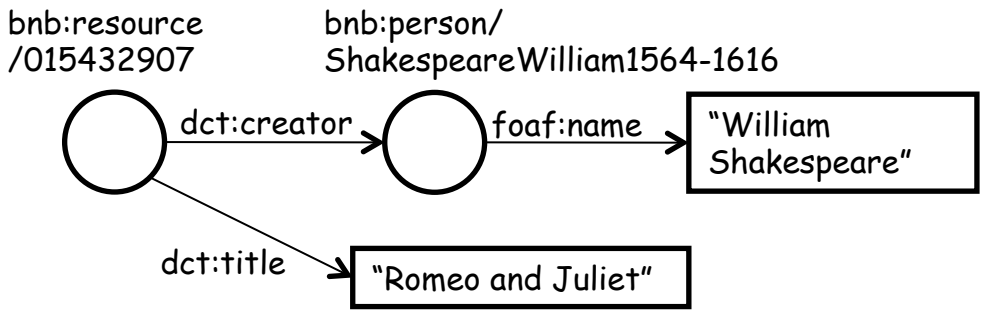
The query returns all resources R for which there are resources X, Y, such that replacing variables ?authors, ?x and ?y, respectively, you get the triples in the queried graph.




**RESULT**

| author |
| --- |
| "Richard Appignanesi" |
| "William Shakespeare" |




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

# SPARQL endpoints

- SPARQL queries are performed 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
    - 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
  - CSV
  - RDF
  - …

# 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 countries with more than 15 millions of inhabitants

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX prop: <http://dbpedia.org/property/>
SELECT ?country_name ?population
WHERE {
        ?country rdfs:label ?country_name.
        ?country prop:populationEstimate ?population .
        FILTER (?population > 15000000 &&
                        langMatches(lang(?country_name), "EN"))
   }
```

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

# SPARQL – example of query on DBPedia

- Return members of rock bands

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

SELECT ?name ?bandname where {
   ?person foaf:name ?name .
   ?band dbo:bandMember ?person .
   ?band dbo:genre dbp:Rock_music .
   ?band foaf:name ?bandname .
}
```

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

# 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> |

# 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-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 `?i10` and `?i13` are not optional. Notice that in this case the from clause has to be removed.

# 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 ?y
WHERE {{?x dct:title "Romeo and Juliet".
        ?x bibo:isbn10 ?i} UNION
       {?x dct:title "Romeo and Juliet".
        ?x bibo:isbn13 ?y}}
```

# SPARQL – Aggregation

*Returns the number of provinces:*

```
PREFIX : <http://My_ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
Select (count(distinct ?x) as ?count)
Where { ?x a :Province. }
```

# SPARQL – Aggregation

*For each province, return the number of cities, but only for those provinces with less than 20 cities:*

```
PREFIX : <http://My_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 :City.
    ?x :belongs_to_province ?p.
    ?p rdfs:label ?provinceName
}
GROUP BY ?provinceName
HAVING (?count <20)
```

# 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}
```

# Graph Databases

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

# 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
  - Universal access mechanism: HTTP
  - Standardized content format: HTML

# 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: Row 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

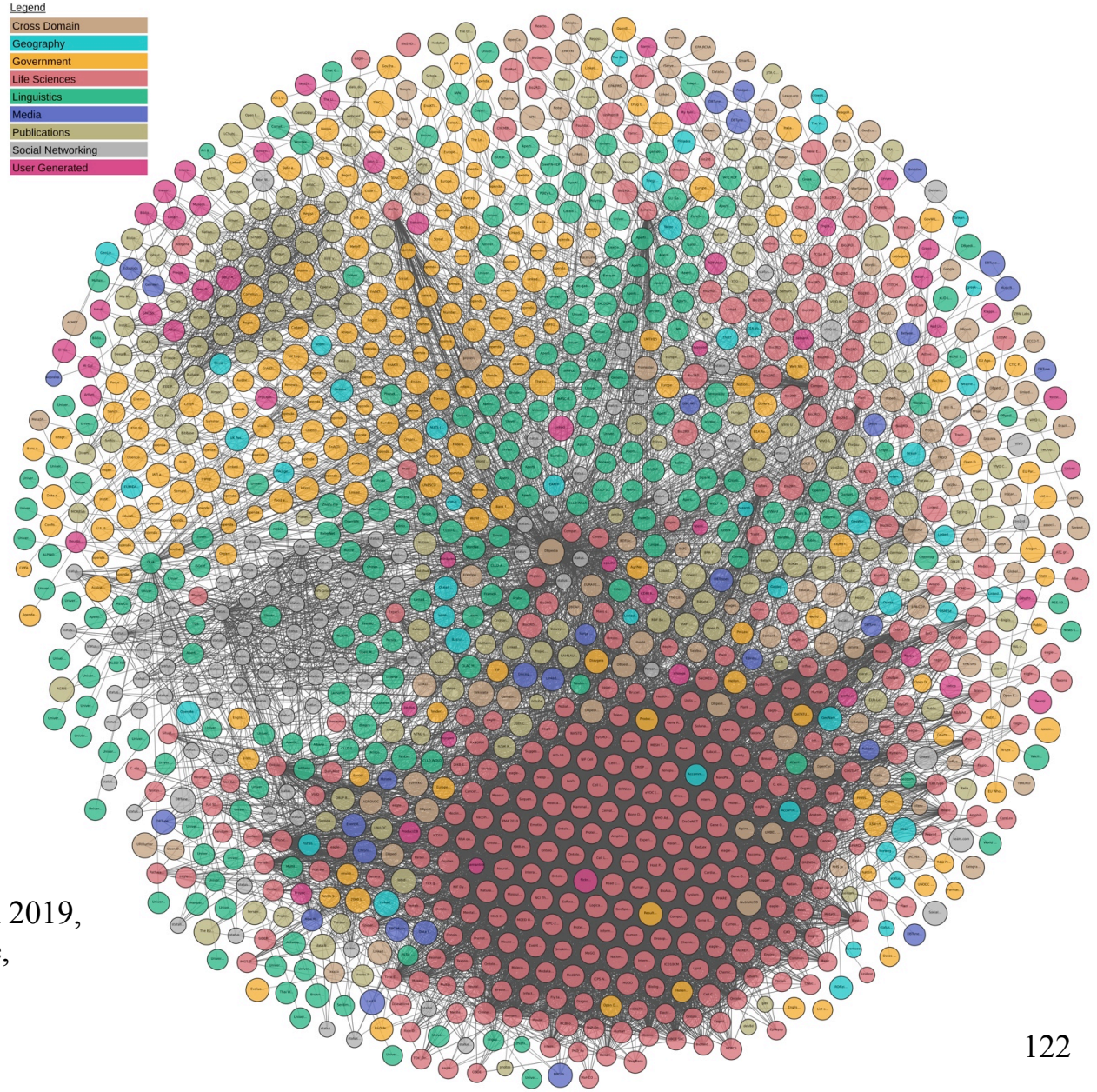
Linked data may be **open** (publicly accessible and reusable) or **closed**

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

http://lod-cloud.net

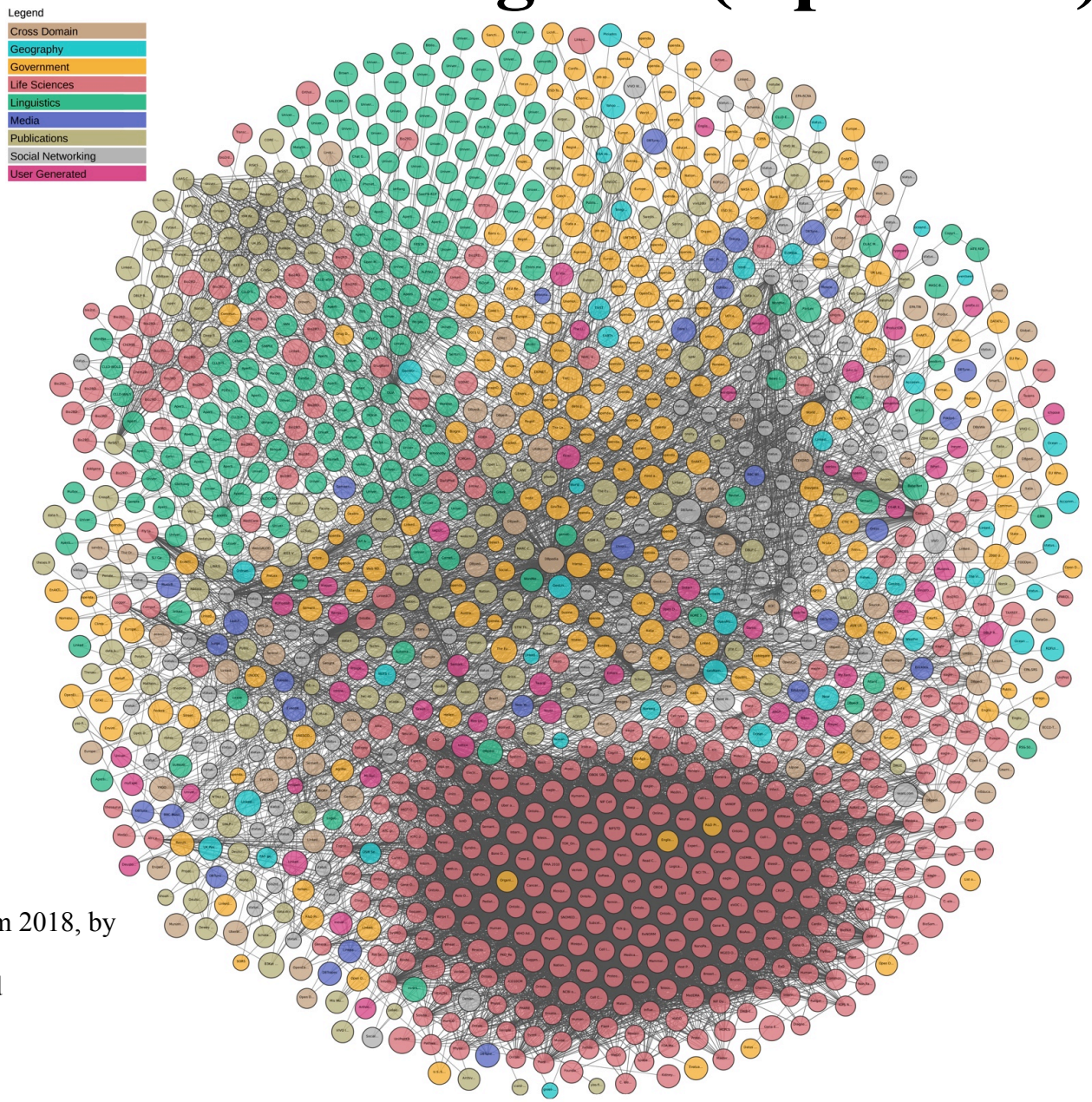
# The LOD cloud diagram (Jan. 2019)

The dataset currently contains 1,234 datasets with 16,136 links



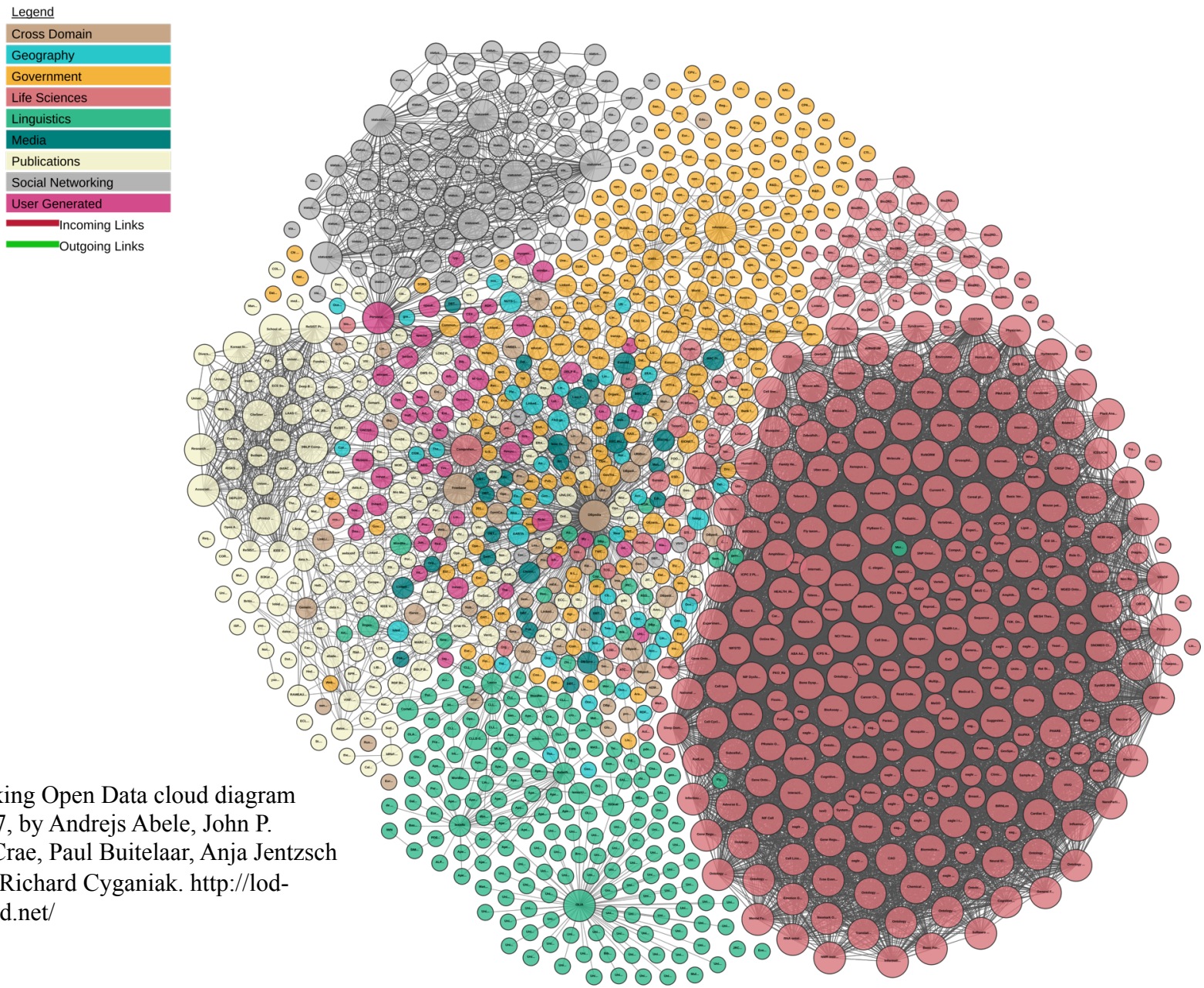

Linking Open Data cloud diagram 2019, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak.
http://lod-cloud.net/

# The LOD cloud diagram (Apr. 2018)



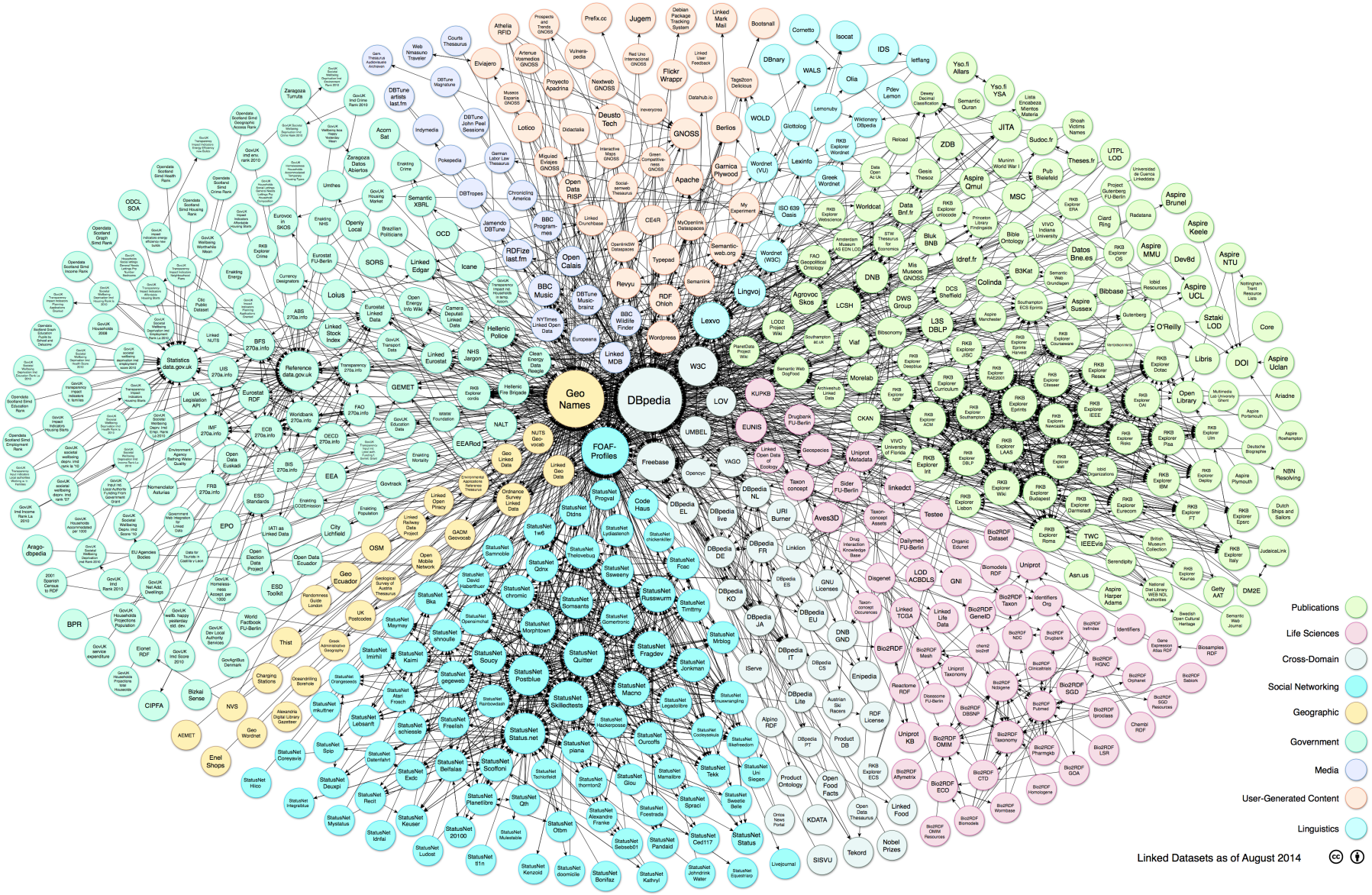

Linking Open Data cloud diagram 2018, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak.
http://lod-cloud.net/

# The LOD cloud diagram (Feb. 2017)



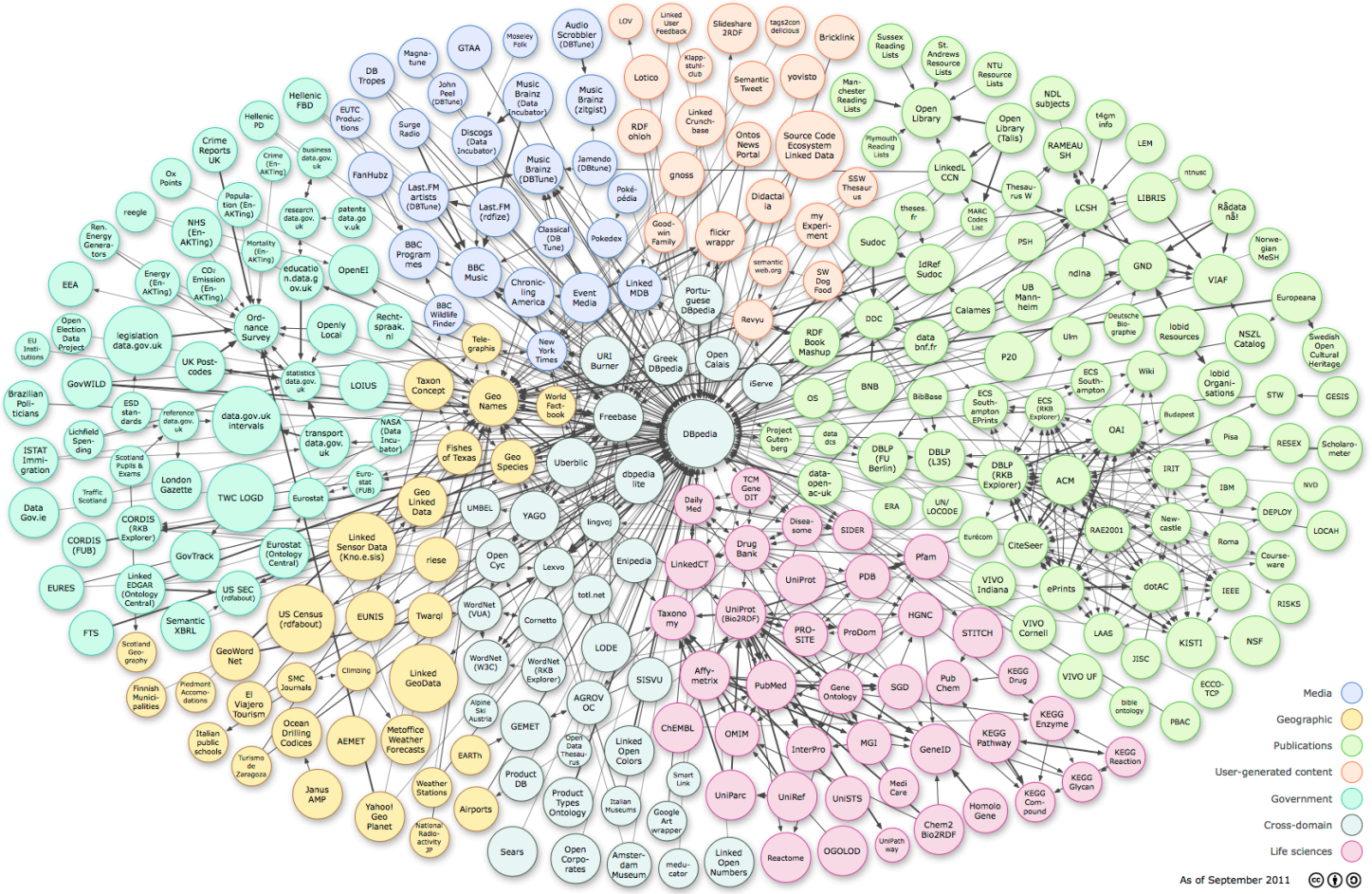

Linking Open Data cloud diagram 2017, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak. http://lod-cloud.net/

# The LOD cloud diagram (Aug. 2014)



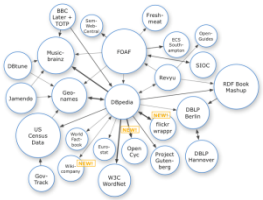

Linking Open Data cloud diagram 2014 M. Schmachtenberg, C. Bizer, A. Jentzsch, R. Cyganiak

# The LOD cloud diagram (Nov. 2011)

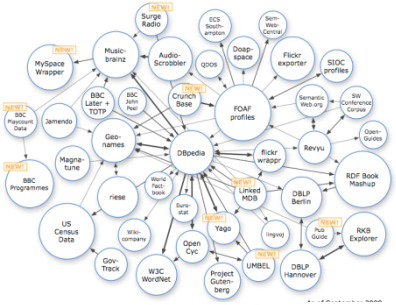



**Linking Open Data** cloud diagram, 09/2011 (by Richard Cyganiak and Anja Jentzsch. http://lod-cloud.net/)

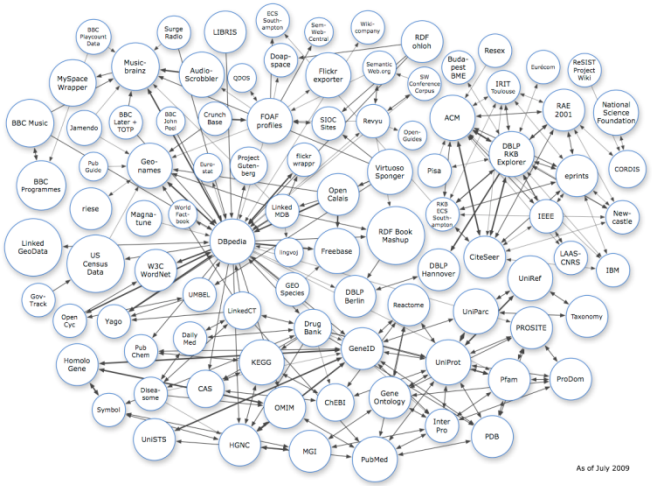
# The LOD cloud diagram



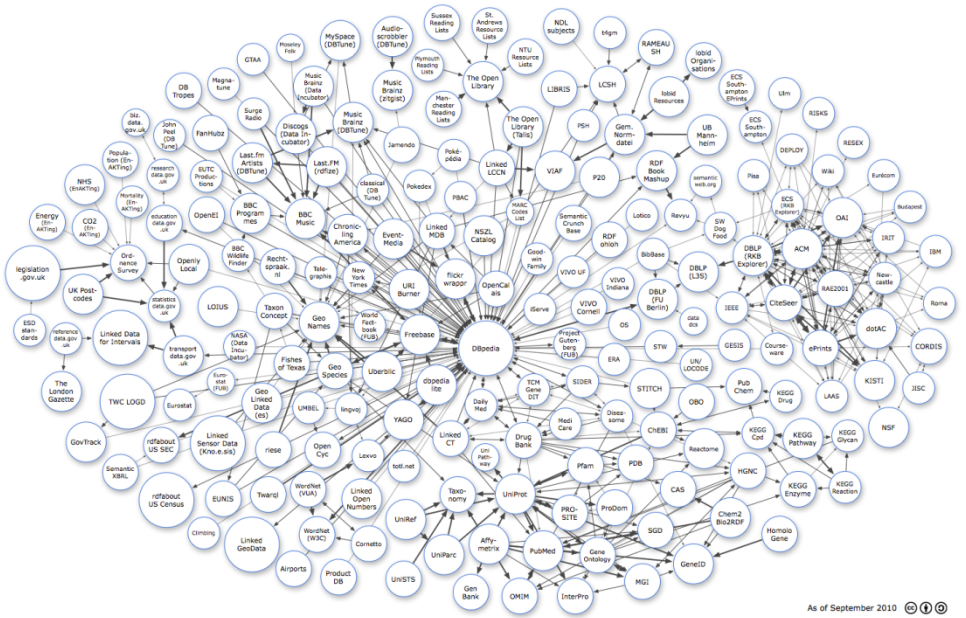
Sep. 2007



Sep. 2008




Sep. 2009




Sep. 2010

# 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/)
- **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

# 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.