BDM-MIRI - Semantic Data Management

Jose Antonio Lorencio Abril

Spring 2023

Professors: Anna Queralt, Óscar Romero

Student e-mail: jose.antonio.lorencio@estudiantat..upc.edu

This is a summary of the course *Semantic Data Management* taught at the Universitat Politècnica de Catalunya by Professors Anna Queralt and Óscar Romero in the academic year 22/23. Most of the content of this document is adapted from the course notes by Abelló and Nadal, [1], so I won't be citing it all the time. Other references will be provided when used.

Contents

List of Figures

List of Tables

List of Algorithms

Part I
Property Graphs

1 Property Graphs

1.1 Definitions

Property graphs were born in the database community, with the idea of enabling the possibility to query and process data in graph form. Up to date, there is not any official standard.

Property graphs are **ocurrence-based**, which means that they are defined by the particular instances inserted, without a pre-existent enforceable schema. The instances are represented by two main constructs:

- **Nodes**: represent entities.
- **Edges**: relate pairs of nodes, and may represent different types of relationships.

Both nodes and edges might be **labeled** and can have a ser of properties, represented as **attributes**[1]. In addition, edges can be defined to be directed or undirected. Moreover, multi-graphs are allowed, meaning that two nodes can be related by multiple edges.

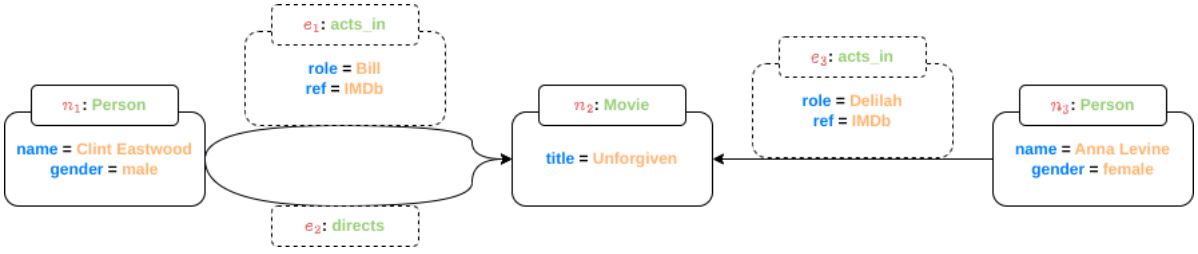
More formally:

Definition 1.1. A **property graph**, G, is a tuple $(V, E, \rho, \lambda, \sigma, Lab, Prop, Val)$, where:

1. V is a finite set of vertices.
2. E is a finite set of edges, such that V and E have no elements in common[a].
3. Lab is a set of labels.
4. $Prop$ is a set of properties and Val is the set of possible values that the properties can take.
5. $\rho : E \to V \times V$ is a total function[b]. Basically, ρ assigns each edge $e \in E$ to the pair of nodes that it relates, $(u, v) \in V \times V$. Usually, $\rho(e) = (u, v)$ means that edge e starts in u and ends in v.
6. $\lambda : (V \cup E) \to Lab$ is a total function. Now, for each vertex and each edge, we assign a label to it[c].
7. $\sigma : (V \cup E) \times Prop \to Val$ is a partial function[d]. Here, we are assigning the values of each property of each node/edge.

[a] Usually, we will identify vertices and edges by their key identifier. Thus, this condition means that a key represents either a node or an edge, but not both at the same time.
[b] A total function is a function $f : Dom \to Im$ such that $\forall x \in Dom, \exists y \in Im | f(x) = y$.
[c] In some definitions, it is possible to assign a set of labels, so that $\lambda : (V \cup E) \to 2^{Lab}$.
[d] A partial function is a function that is not necessarily total.

Example 1.1. A simple graph.



In this example, we have the visual representation of a simple graph. Let's create each of the components of the formal definition:

[1] In some definitions, edges are not allowed to have attributes.

- $V = \{n_1, n_2, n_3\}$.
- $E = \{e_1, e_2, e_3\}$.
- $Lab = \{Person, Movie, acts\_in, directs\}$.
- $Prop = \{name, gender, title, role, ref\}$.
- $Val = \{Clint\ Eastwood, Anna\ Levine, male, female, Unforgiven, Bill, Delilah, IMDb\}$.
- $\lambda(n_1) = Person, \lambda(n_2) = Movie, \lambda(n_3) = Person, \lambda(e_1) = acts\_in, \lambda(e_2) = directs, \lambda(e_3) = acts\_in$.
- $\sigma(n_1, name) = Clint\ Eastwood, \sigma(n_1, gender) = male, \sigma(n_2, title) = Unforgiven, \sigma(n_3, name) = Anna\ Levine, \sigma(n_3, gender) = female, \sigma(e_1, role) = Bill, \sigma(e_1, ref) = IMDb, \sigma(e_3, role) = Delilah, \sigma(e_3, ref) = IMDb$.

1.2 Traversal Navigation

The **graph traversal pattern** is defined as the ability to rapidly traverse structures to an arbitrary depth and with an arbitrary path description.

This framework is totally opposite to set theory, on which relational databases are based on. In the relational theory, this is equivalent to joining data and selecting data, while in a graph database, the relationships are explicit (there no foreign keys), there is no need to add nodes for artificial concepts and we can consider the joins as being hard-wired in the data. This makes traversing from one node to another a constant time operation.

Traversing graph data depends on three main variables:

- The **query topology**, which refers to the complexity of what are we looking for, and the **traversal seeds**, which are the way in which the search is started.
- The **size of the graph**, typically measured as the number of edges.
- The **topology of the graph**.

1.3 Graph operations

There are basically two types of operations:

- **Content-based queries**: in these queries, the value is relevant. We want to get a specific node, or the value of some attributes of a node/edge, etc. For example, aggregations.
- **Topological queries**: in this case, only the topology of the graph is considered. Several business problems are solved using graph algorithms exploring the graph topology. For example, computing the betweenness centrality of a node.
- **Hybrid approaches**: leverage both types of queries.

Of these types, we are going to focus on topological queries.

1.3.1 Topological queries

Adjacency queries

Two nodes are adjacent if there is an edge between them (usually disregarding direction). Therefore, the adjacency of a node is defined as all nodes adjacent to it:

$$Adjacency(n) = N | n_i \in N \iff \exists e : \rho(e) = (n_i, n) \vee \rho(e) = (n, n_i).$$

The computational cost of this operation is linear on the number of edges to visit.

Examples of use cases are finding all friends of a person, airports with direct connection,...

Reachability queries

A node is reachable from another node if there is a set of edges, called a **walk**, that can be traversed to get from one to the other (in this case, direction is usually taken into account, but it could be disregarded if needed). Now, the definition is as follows:

$$Reachability\left(n_{or}, n_{dest}\right) = True \iff$$

$$\exists Walk\left(n_{or}, n_{dest}\right) = \left(e_1, ..., e_m\right) | \exists n_1, ..., n_{m-1} : \rho\left(e_1\right) = \left(n_{or}, n_1\right), \rho\left(e_2\right) = \left(n_1, n_2\right), ..., \rho\left(e_m\right) = \left(n_{m-1}, n_{dest}\right).$$

Additional constraints can be defined:

- Fixed-length paths: we can fix the number of edges and nodes of the walk.
- Shortest path: find the walk that minimizes some metric, such as the number of hops or the sum of weights of the edges,...
- Non-repeated nodes: in this case the walk is called a **path**.
- Non-repeated edges: in this case the walk is called a **trail**. Note that a path is more restrictive than a trail.
- Regular simple paths: we can restrict the path to respect some regular expression.

The computational cost is high for large graphs, and it also depends on what constraints are we imposing. For instance, if we want to compute the shortest path, we can use **Dijkstra's algorithm**, which is $O\left(|V|^2\right)$, or $O\left(|E| \cdot |V| \log |V|\right)$ using priority queues.

Examples of use cases are finding all friends of a friend, all flight connections,...

Label-constrained reachability

We compute reachability, imposing that all edges in the walk have a label in a defined set of labels:

$$G_L^* = \left\{(s,t) | \exists p \in paths\left(s,t\right) \ such \ that \ \lambda\left(e\right) \in L, \forall e \in p\right\}.$$

Another way to see this is that we are trying to determine all pairs of nodes such that there is a path between them such that the concatenation of the edge labels along the path forms a string in the language denoted by the regular expresion $\left(l_1 \cup ... \cup l_n\right)^*$, where $L = \{l_1, ..., l_n\}$.

Typically, the allowed topology and labels involved are expressed as a regular expression. In general, this problem is known to be NP-complete.

Pattern matching

In this case, we want to find all subgraphs that follow a given pattern. More formally, we have $G = (V, E)$ and a pattern $P = (V_p, E_p)$ and we want to find all $G' = (V', E')$ such that $V' \subset V, E' \subset E$ and $P \cong G'$, i.e., P and G' are isomorphic, i.e., there are biyections $V' \xrightarrow{f_1} V_p$ and $E' \xrightarrow{f_2} E_p$.

This problem is also NP-complete.

Examples of use cases are finding all groups of cities such that all of them are directly connected by flights (find cliques),...

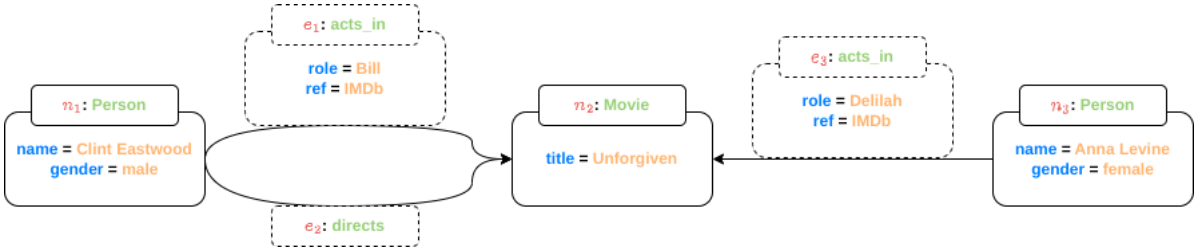
1.4 Property Graph Patterns

Among the operations that we have seen so far, it is interesting, in the context of property graphs, to focus on pattern matching. Now, we use **basic graph patterns (bgps)**, which are equivalent to conjunctive queries, and are a property graph where variables can appear in place of any constant.

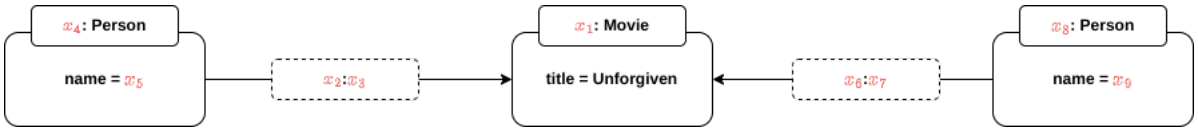
A **match** for a bgp is a mapping from variables to constants, such that when the mapping is applied to the bgp, the result is a subgraph of the original graph.

The **results** for a bgp are all mappings from variables in the query to constants that comprise a match.

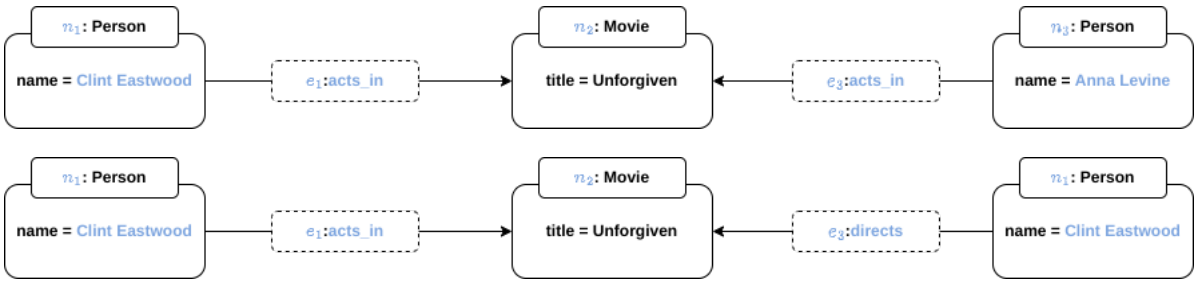
Example 1.2. A simple pattern matching. Assume we have the same graph we used before:



And the following bgp:



Here, I have coloured variables in red, and left constant in black.
Let's see some matches:



And so on...

1.4.1 Evaluating graph patterns

Now, we are going to formalize a bit the intuition built in the previous explanation and example. Evaluating a bgp P, against a graph G corresponds to listing all possible matches of P with respect to G:

Definition 1.2. Given an edge-labelled graph $G = (V, E)$ and a bgp $P = (V', E')$, a **match** of P in G is a mapping

$$h : Const \cup Var \to Const$$

such that:

1. For each constant $a \in Const$, $h(a) = a$, i.e., constants are preserved.
2. For each edge $(b, l, c) \in E'$, we have $(h(b), h(l), h(c)) \in E$. This imposes that
   (a) Each edge of P is mapped to an edge of G.
   (b) The structure of P is preserved in its image under h in G.

1.4.2 Semantics of a match

Matches can be defined using different semantics on what we consider equivalent graphs, and what conditions the function h have to meet:

- **Homomorphism-based semantics**: multiple variables in P can map to the same constant in G (h is not necessarily inyective). This correspondes to the familiar semantics of select-from-where queries in relational databases.
- **Isomorphism-based queries**: we add the constraint that h must be inyective.

Nonetheless, there are intermediate solutions:

- **Strict isomorphism**: corresponds to the isomorphism-based queries, in its stricter sense. h is inyective.
- **No repeated-node semantics**: h is only inyective for nodes.
- **No repeated-edge semantics**: h is only inyective for edges.

2 Graph Query Languages

Graph Query Languages are declarative languages used to query a graph. Typically, a GQL matches an extended version of pattern matching, and each database engine chooses fix semantics for it, not existing a common agreement nor standard. There are also APIs provigin implementation of graph metrica or label-constrained shortest path, which, depending on the metric or algorithm chosen, maps to adjacency, reachability or pattern matching.

2.1 Types of queries

There are different types of queries, each of them using a different access plan:

- **Adjacency queries**: neighbourhood queries require accessing the basic data structure and navigate it. Thus, their performance depends on the database implementation and the specific query, as the time to find a node or edge depends on this implementation.
- **Regular path queries**: combine pattern matching and reachability and require specific graph-oriented algorithms. They are equivalent to conjunctive queries. They are also called navigational graph patterns.

  RPQs extend the bgp definition by allowing regular expressions on edges to describe path queries in a pattern, i.e., a **path** is described as

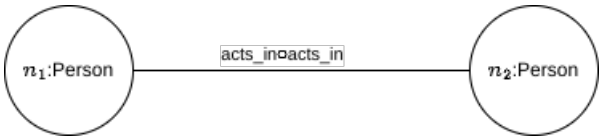
  $$x \xrightarrow{\alpha} y \text{ over } G,$$

  where x, y are nodes in G and α is a regular expression over Lab. The regular expressions differ from language to language. Some usual expressions are:

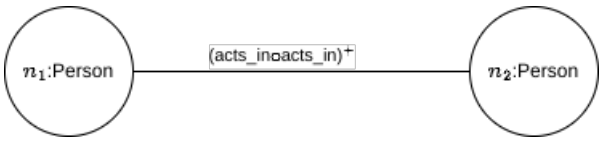
  - The Kleene star *: 0 or more ocurrences.
  - The Kleene plus +: 1 or more ocurrences.
  - Concatenation $\circ$
  - Inverse $^-$
  - Union |:
  - Combinations of them and the labels in Lab.

  Example 2.1. Some simple RPQs. In our previous example, we can define some simple RPQs:

  - Find all co-actors of all actors:

  - Retrieve all actors you can reach by transitively following the co-actoring relationship, at least once:

- **Complex graph patterns**: add further expressivity beyond conjunctive queries, such as groupings, aggregations and set operations. The previous RPQs are equivalent to conjunctive queries without projections, but database languages are richer than this, enforcing GQLs to implement more complex semantics. **GraphQL** was the first graph algebra extending RPQs with relational-like operators.

2.2 Popular languages

2.2.1 Cypher

Cypher was created by Neo4j, and acts as a de facto standard, adopted by other graph databases. It is a high-level, declarative language, providing both DDL (Data Definition Language) and DML (Data Modification Language) capabilities, and allowing navigational graph patterns, except concatenation.

It applies pattern matching under no-repeated-edges isomorphism semantics. The available clauses are:

- DML:
  - MATCH: the graph pattern to match.
  - WHERE: filtering criteria.
  - WITH: divides a query into multiple distinct parts.
  - RETURN: define what to return.
- DDL:
  - CREATE | MERGE: creates nodes and relationship. Merge does it only if it does not exist beforehand, entailing an overhead.
  - DELETE: removes nodes, relationships and properties.
  - SET: set values of properties.
  - FOREACH: performs updating actions once per element in a list.

Cypher applies a data pipeline, where each stage is a MATCH-WHERE-WITH/RETURN block, allowing the definition of aliases to be passed between stages.

For example, uppose you have a graph database with `Person` nodes and two types of relationships: `FRIENDS_WITH` and `WORKS_WITH`. You want to find all mutual friends of `Alice` and `Bob` who also work with someone named `Charlie`. You can achieve this with the following Cypher query:

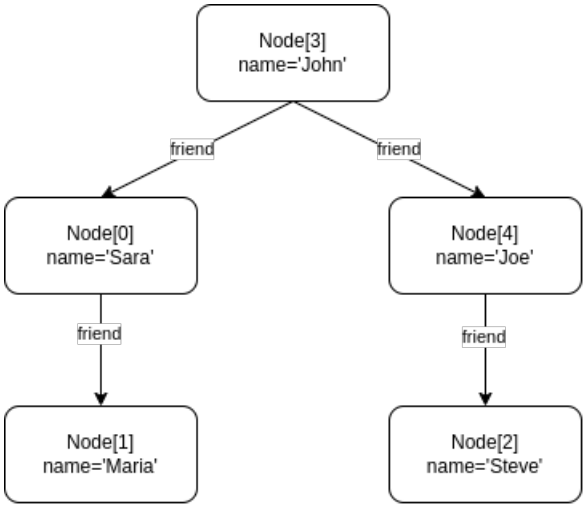
```
MATCH (alice:Person {name: 'Alice'})-[:FRIENDS_WITH]-(mutual_friend:Person)-[:FRIENDS_WITH]-(
    bob:Person {name: 'Bob'})
WITH mutual_friend
MATCH (mutual_friend)-[:WORKS_WITH]-(charlie:Person {name: 'Charlie'})
RETURN mutual_friend.name
```

In this query, we have three stages in the data pipeline:

1. MATCH (Stage 1): Match the graph pattern where `Alice` and `Bob` have mutual friends. Bind the `mutual_friend` node.
2. WITH: Pass the `mutual_friend` node to the next stage.
3. MATCH (Stage 2): Match the graph pattern where the `mutual_friend` from the previous stage works with `Charlie`.
4. RETURN: Return the name of the mutual friend who meets both criteria (being a mutual friend of `Alice` and `Bob` and working with `Charlie`).

In this example, the `WITH` clause is used to separate the query into two stages. The first stage finds all mutual friends of `Alice` and `Bob`, and the second stage filters those mutual friends to only include the ones who work with `Charlie`. The use of `WITH` here is what enables the pipelining of stages in the query.

Example 2.2. Given the following graph:



1. Return all nodes

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

2. Return all edges

```
MATCH ()-[e]-()
RETURN e;
```

3. Return all neighbour nodes of 'John'

```
MATCH (john {name:'John'})-[:friend]-(f)
RETURN john, f;
```

4. Return the incident nodes of all edges

```
MATCH (n1)-[e]->(n2)
RETURN e, n1, n2;
```

2.2.2 GQL

There is an ongoing big effort towards stardardization of Graph Query Languages, through the GQL project.

3 Graph Processing

Exercise 3.1. Understand the relationships between the basic graph operations. Answer the following questions:

1. Is adjacency subsumed by reachability?

   No, because there is no way to compute all adjacent nodes to a given node by answering reachability queries.

2. Is adjacency subsumed by pattern matching?

   Yes, the adjacency query $Adjacent\,(x)$ is equivalent to the pattern matching query $Match\,(x \to y)$.

3. Is reachability subsumed by pattern matching?

   Yes, the reachability query $Reachable\,(x, y)$ is equivalent to $!not\_empty\,(Match\,(x \to^* y))$.

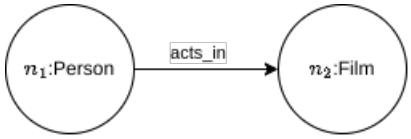
Note that in the cases in which a query is subsumed in another, it makes sense to have the simplified version, which can be optimized as a simpler case of pattern matching.

Notice that so far, the operations have been presented conceptually, being agnostic of the underlying technology. The theoretical costs are:

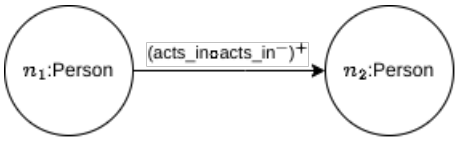
- Adjacency is linear in the amount of vertices to visit $O(|V|)$.
- Reachability is $O\left(|V|^3\right)$ using Dijkstra's shortest path.
- Label-constrained reachability is $O(|V|)$ for a single pair of vertices and $O\left(|V|^3\right)$ for all pairs. It is NP-complete if we enforce no-repeated-edges isomorphism semantics.
- Pattern matching in general is NP-complete.
- Navigational pattern matching is also NP-complete, but can be reduced to $O\left(|V|^3\right)$ using bounded simulation algorithms.

Exercise 3.2. Identify the most efficient algorithm to solve a given query. Assume a graph containing relationships and nodes of actors and films (the same as before, but with virtually more information). Define:

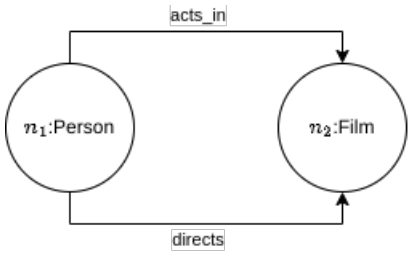
1. A query that should be solved as an Adjacency problem.
   Retrieve all pairs of $(Person, Film)$ such that $Person$ acts in $Film$.



2. A query that should be solved as a Label-constrained Reachability problem.
   Retrieve all pairs of actors related by coacting relationship.



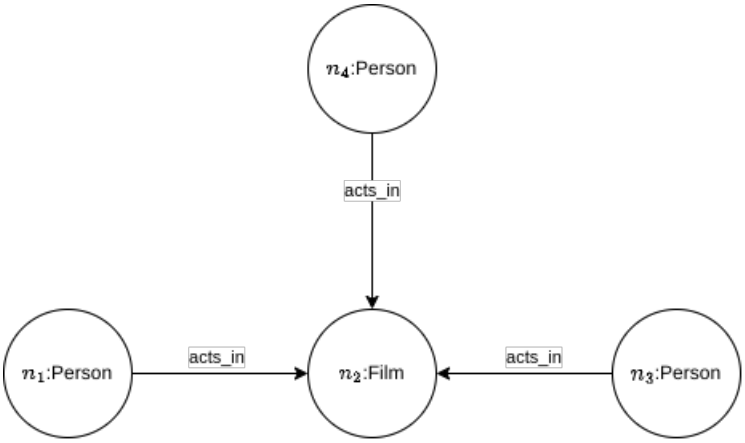
3. A query that should be solved as a Navigational Pattern matching problem
   People and movies in which the person acts and directs.



   NO! This can be done with reachability, repeating the same node:

   So, we need something else... If we fix something in the middle, then we cannot use reachability. For example, retrieve all co-actors related by the Movie Titanic:

We can also enforce other complex constraints, such as: retrieve all movies in which there are exactly 3 actors:



3.1 Dijkstra's algorithm

Dijkstra's algorithm defines a method to find the shortest path between two nodes in a graph in which the edges have a cost assigned (it can be used in general if we take the distance between two nodes as the number of edges used to go from one to the other).

More formally, if we have a path between two nodes, u, v, $P(u, v) = \{e_1, ..., e_n\}$, then the distance of the path is

$$d(u, v) = d(P) = \sum_{i=1}^{n} c(e_i),$$

where $c(e_i)$ is the cost of going through edge e_i. If we want to account only for the number of edges used, we can then define $c(e) = 1$ for all $e \in E$.

Dijkstra[2] noticed two very convenient properties of shortest paths:

1. Every subpath of a shortest path is itself a shortest path.

   This is easy to see, since if we have a shortest path between u and v, $P(u, v)$, and we take two of the nodes that are included in the path, say u_1 and u_2, then the subpath that goes from u_1 to u_2 must also be of shortest length. Imagine it was otherwise, then there would be a path $P'(u_1, u_2)$ shorter than the subpath we found in P. Then, we could substitute this subpath by P', and the new path would be shorter than P. But P is a shortest path, so this is not possible.

2. The triangle inequality of shortest paths:

$$d(u, v) \leq d(u, x) + d(x, v), \ \forall x \in E.$$

The algorithm is detailed in Algorithm 1. Note that this algorithm is a bit more general, since it allows to find the shortest paths from one source node U to the rest of the nodes in the graph. If we want a specific one, we can stop when we reach it or just find it from the output. To get the shortest path from U to V, we would go to $prev[V]$ and traverse this in reverse until getting to U.

3.2 Pattern Matching

Even considering the most basic fragment of graph patterns and for all semantics applied (homomorphism, isomorphism, no-repeated node isomorphism or no-repeated edge isomorphism), the problem is NP-complete, and the problem is tackled using different techniques, algorithms and heuristics.

[2] See Dijkstra on Wikipedia.

```
for each vertex v in G.Vertices:
  dist[v] <- INFINITY
  prev[v] <- UNDEFINED
  add v to Q
dist[U] <- 0

while Q is not empty:
  u <- vertex in Q with min dist[u]
  remove u from Q

  for each neighbor v of u still in Q:
      alt <- dist[u] + Graph.Edges(u, v)
      if alt < dist[v]:
          dist[v] <- alt
          prev[v] <- u

return dist[], prev[]
```

Algorithm 1: Dijkstra(WeightedGraph G, SourceNode U)

3.3 Complex Graph Processing

3.3.1 Graph Metrics

A **metric** can be defined as a combination of adjacency, reachability, pattern matching and complex graph patterns, so the cost of a metric depends on how it is defined. Nevertheless, there are very usual and relevant metrics, which are typically provided as built-in functions. For example, the min/max degree in the graph, its diameter, pageRank,...

3.3.2 Graph Processing Pipelines

A pipeline is a list of algorithms over a graph, which are pipelined, inputting the output of an algorithm to the next one, to obtain some result. Therefore, we can see a metric as a pre-defined pipeline, but a pipeline can be as complex as we want/need.

3.3.3 Graph Embeddings

An embedding is a vector representation of a graph, and it is useful to perform data analysis using typical ML algorithms, that have been developed using vectors.

4 Graph Databases

A **graph database** is a software that provides a way to store and persist graph data, as well as means to process this data. Examples of graph databases are Neo4j or Titan.

A **distributed graph framework** refers to a processing framework over a graph database. We could compare it to MapReduce[3] or to Spark[4], in the sense that it is a mean to extract data from a graph database, but not to store graphs. Examples of these frameworks are Pregel or Giraph.

4.1 Implementation of Graph Databases

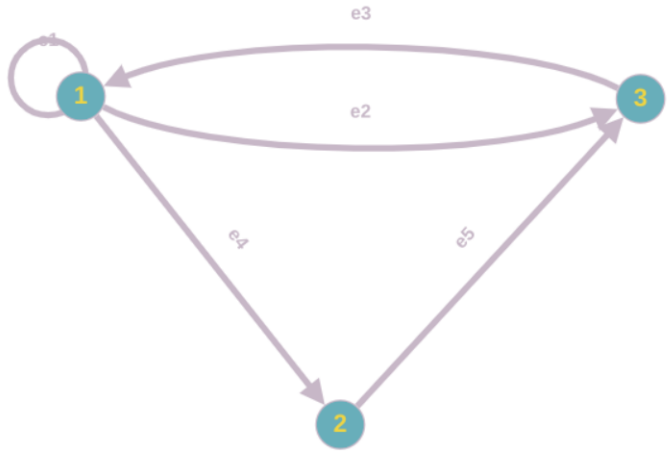
Graph databases can be implemented in different ways, and each approach presents advantages and disadvantages.

4.1.1 Incidence Lists

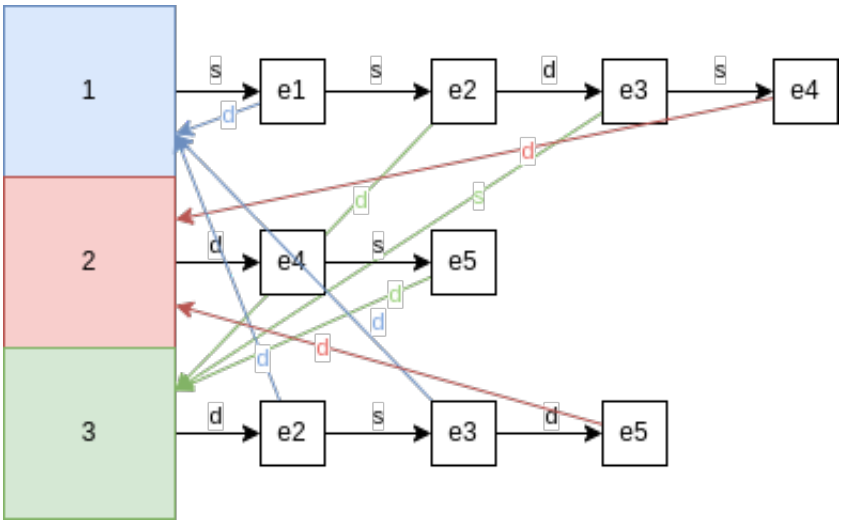
In incidence lists, vertices and edges are stored as records of objects, such that each vertex stores incident edges, and each edge stores incident nodes.

Example 4.1. An Incidence List

This graph:



Can be encoded in an incidence list as follows:



The diagram is a bit messy, sorry for that.

[3] See MapReduce in Wikipedia.
[4] See Apache Spark in Wikipedia.

Incidence Lists in Neo4j

Neo4j is implemented using incident lists, using the concept of linked lists.

Nodes

In Neo4j, there is one physical file to store all nodes in-memory, with a Least Frequently Used cache policy, and a fixed size for the records (of 15B). This enables for very fast look-ups in $O(1)$ time.

Let's delve into this. This is the anatomy of a Node:

inUse	nextRelId	nextPropId	labels	extra
1B	4B	4B	5B	1B

The first byte is used for some metadata, such as a flag 'inUse'.
The bytes 2-5 are the ID of the first relationship (edge) incident to the node.
The bytes 6-9 are the ID of the first property of the node.
The bytes 10-14 encode the labels of the node.
The last byte contain extra information.

Relationships and properties

There are two more kind of files to encode relationships and properties, also containing fixed size records and using a LFU cache policy. A relationship looks like this:

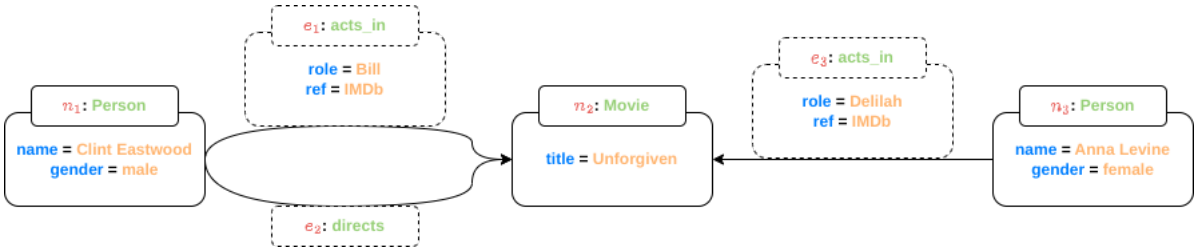
meta	sNodeId	dNodeId	typeId	sNodePrevRelId	sNodeNextRelId	dNodePrevRelId	dNodeNextRelId	nextPropId
1B	4B	4B	4B	4B	4B	4B	4B	4B

The property file looks like this:

meta	node/edgeId	prevPropId	nextPropId	propNameId	propValueId
1B	4B	4B	4B	4B	4B

In this case, the metadata field has a bit indicating whether the property belongs to a node or a relationship.

Example 4.2. Encode this graph using the seen nomenclature:



Nodes:

	meta	firstRel	firstProp	labels	extra
n1		e1	p1	l1	
n2		e1	p5	l2	
n3		e3	p6	l2	

Relationships:

	meta	sId	dId	label	sNprevR	sNnextR	dNprevR	dNnextR	prop
e1		n1	n2	l3	-	e2	-	e2	p3
e2		n1	n2	l4	e1	-	e1	e3	-
e3		n3	n2	l3	-	-	e2	-	p8

Properties:

	meta	n/eId	prevP	nextP	idName	idValue
p1	n	n1	-	p2	na1	v1
p2	n	n1	p1	-	na2	v2
p3	e	e1	-	p4	na3	v3
p4	e	e1	p3	-	na4	v4
p5	n	n2	-	-	na5	v5
p6	n	n3	-	p7	na1	v7
p7	n	n3	p6	-	na2	v8
p8	e	e3	-	p9	na3	v6
p9	e	e3	p8	-	na4	v4

Names:

na1	name
na2	gender
na3	role
na4	ref
na5	title

Values:

v1	Clint Eastwood
v2	Male
v3	Bill
v4	IMDb
v5	Unforgiven
v6	Delilah
v7	Anna Levine
v8	Female

Labels:

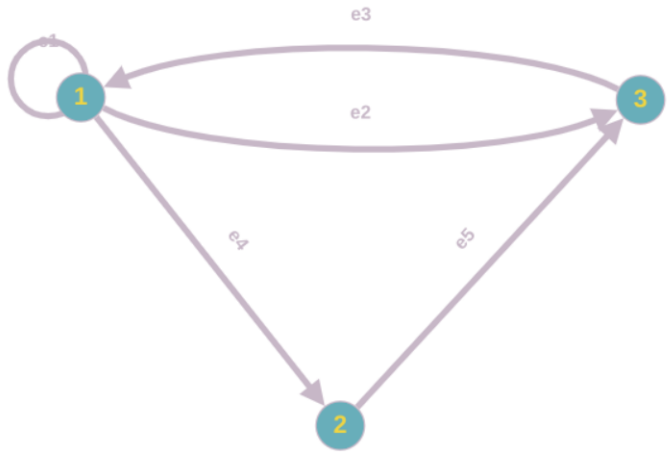
l1	Person
l2	Movie
l3	acts_in
l4	directs

4.1.2 Adjacency Lists

For each node, we store the list of its neighbors. If the graph is directed, the list contains only the outgoing nodes. This approach makes it cheaper for obtaining the neighbors of a node, but it is not suitable for checking if there is an edge between two nodes, since we would need to travers the lists for both nodes completely.

Example 4.3. An Adjacency List

This graph:



Can be encoded in an adjacency list as follows:

[1] → [1] → [2] → [3]

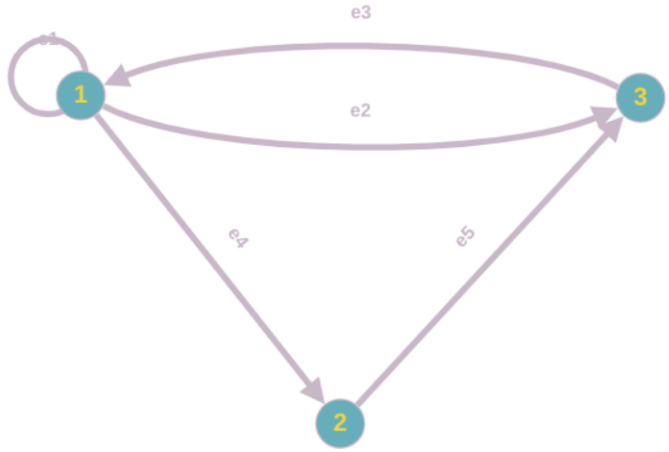
[2] → [3]

[3] → [1]

4.1.3 Incidence Matrix

An incidence matrix is a bidimensional graph representation, in which rows represent vertices and columns represent edges. Then, a non-null entry represents that the source vertex is incident to the edge.

Example 4.4. An Incidence Matrix
This graph:



Can be encoded in an incidence matrix as follows:

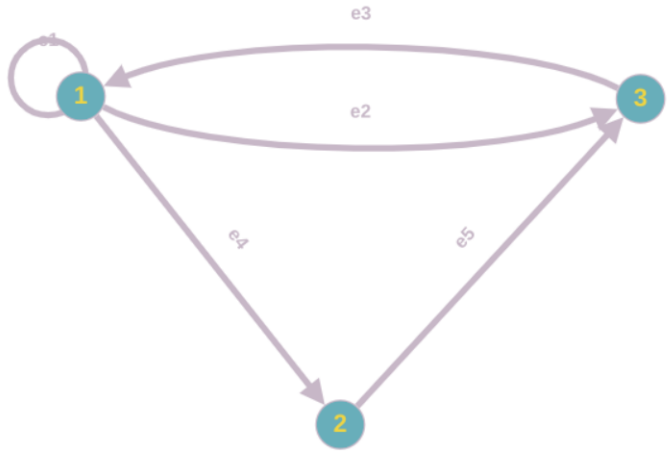
		edges				
		e1	**e2**	**e3**	**e4**	**e5**
nodes	**1**	3	1	2	1	0
	2	0	0	0	2	1
	3	0	2	1	0	2

Where 0='not incident', 1='source', 2='dest' and 3='source&dest'.

4.1.4 Adjacency matrix

This is also a bidimensional graph representation, in which rows represent source vertices, and columns represent destination vertices. Therefore, each non-null entry represents that there is an edge from the source node to the destination node.

Example 4.5. An Adjacency Matrix
This graph:



Can be encoded in an adjacency matrix as follows:

		destination		
		1	**2**	**3**
source	**1**	1	1	1
	2	0	0	1
	3	1	0	0

4.2 Types of Graph Databases

Some graph databases and graph processing frameworks are based on strong assumptions that are not always explicit, but are rather a consequence of the internal implementation of graphs. We can distinguish between:

- **Operational graphs**: they are the graph equivalent of a CRUD database. In this kind of graph database, nodes and edges can be deleted, updated, inserted and read. Examples are Neo4j or OrientDB.
- **Analytical graphs**: these are snapshot graphs that cannot be modified by the final user, so they are the equivalent of a data warehouse. For example, the graph processing frameworks can be seen as analytical graphs.

5 Distributed Graph Processing

When we have a centralized graph, the cost of the queries that we launch on it depends on the number of edges/nodes visited during processing. Therefore, this cost is affected by the graph size and its topology, and the processing algorithm used. But sometimes the graph is large and the algorithm expensive. For instance, navigational pattern matching, in the best case, is still of cubic computational complexity.

Also, graph computations are difficult to scale and parallelize, because:

- Computations are data-driven: this means that the computations are driven by vertices and edges, with the structure of the computation not known a priori.
- Unstructured problems: the data stored in graphs is usually unstructured and irregular, making it difficult to partition.
- Poor locality: the computations and access patterns tend not to have very much locality.
- High data access to computation ratio: since exploring the structure of a graph is more usual than performing large numbers of computations.

Sequential graph algorithms, which require random access to all the data, present poor locality and together with the indivisibility of the graph structure cause time and resource intensive pointer chasing between storage mediums in order to access each datum. In response to these shortcomings, new distributed frameworks based on the vertex-centric programming model were developed. This approach is:

- No shared-memory (there is a local view of data).
- Meant to converge upon iteration.
- Naturally adapting to distributed settings.

As opposed to having a global perspective of the data (assuming all data is randomly accessible in memory), vertex-centric frameworks employ a local, vertex oriented perspective of computation, introducing the paradigm **Think Like A Vertex (TLAV)**.

5.1 Distributed Graph Storage

Several open-source solutions like HDFS, HBase, or Apache Titan can be used for storage. Proprietary solutions like Amazon Neptune also exist. Each approach has its own trade-offs: open-source solutions offer flexibility but may demand additional maintenance and expertise, while proprietary solutions provide comprehensive support but may pose usage limitations and costs.

For distributed processing frameworks to function effectively, graph data needs to be exposed as two views: a set of vertices and a set of edges. Traditional distributed data management considerations, like partitioning and replicas, apply to these views.

5.2 TLAV Frameworks

TLAV frameworks operate on a message passing interface and support iterative execution of a user-defined vertex program. Vertices pass messages to adjacent vertices, and this iterative process continues until a termination condition is met.

They might either follow the **Bulk Synchronous Parallel (BSP) computing model**, in which computation is based on **superstep**, where a superstep must finish entirely before the next superstep starts, defining a synchronization barrier per superstep; or an **asynchronous computing model**, which is prone to suffer from deadlocks and data races, but may improve the performance under certain assumptions/conditions.

Examples of TLAV frameworks include Pregel, Apache Giraph, and GraphX, each offering its own strengths and weaknesses. It's also important to note the role of fault-tolerance in distributed graph processing. Ensuring system resilience to node failures is a critical aspect that helps maintain operation continuity.

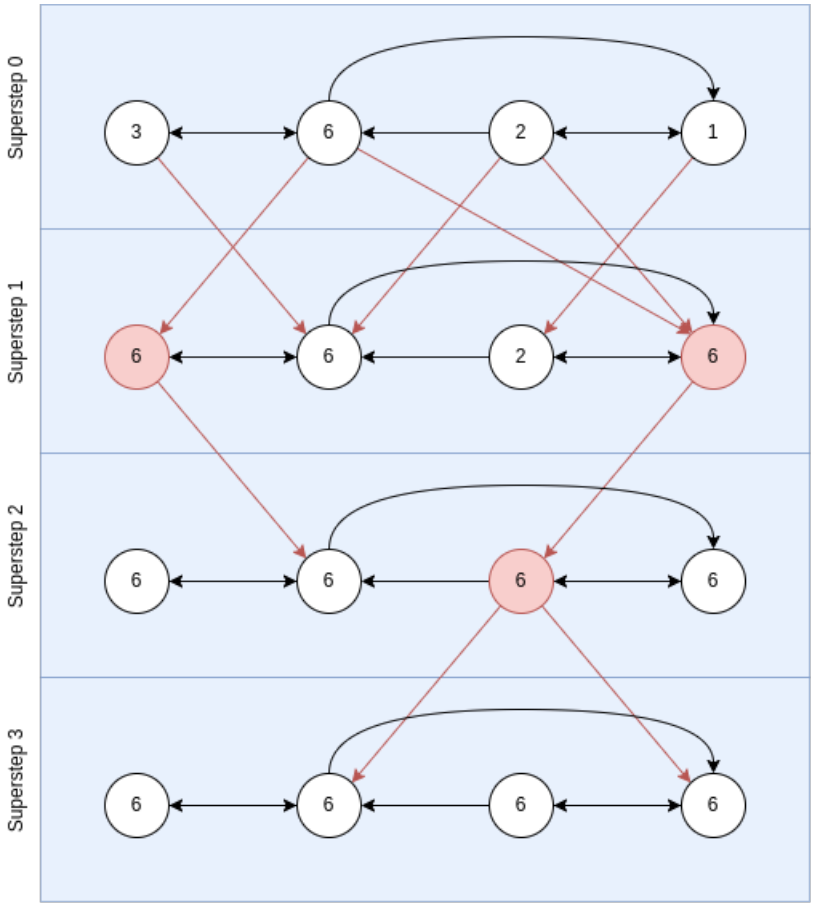
Example 5.1. Calculating max using TLAV.

In the first superstep: all vertices send its value to its adjacent vertices.

On each superstep: each vertex compares the value that it has received (if any) to the current value that it has. If it is greater, then it updates. In case of update, it sends again this new value to adjacent nodes.

Stop condition: no vertex changes in a superstep.

The process could be as the following:

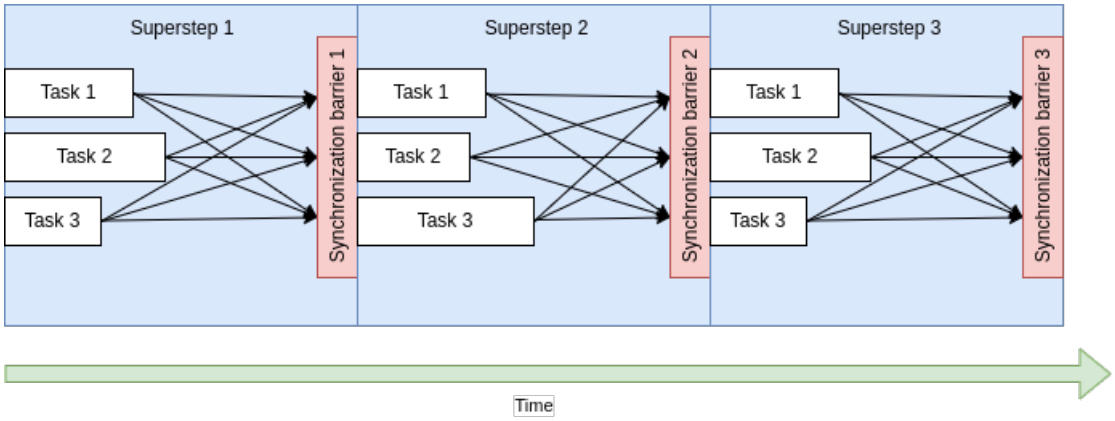


Here, when a node is red, is because it updated, and the red arrows indicate messages. The process finishes when no node changes.

5.2.1 Synchronized TLAV

As we have explained, TLAV framework supports iterative execution of a user defined vertex program over vertices of the graph. Programs are thus composed of several interdependent components that drive program execution, the **vertex kernels**. A **synchronization barrier** is set between interdependent components, defining the supersteps.

Therefore, a superstep is composed of different kernels, and it ends when all kernels finish, and all messages are sent. Then, there is a synchronization barrier, which is used to synchronize the obtained results, so that the next superstep can begin.

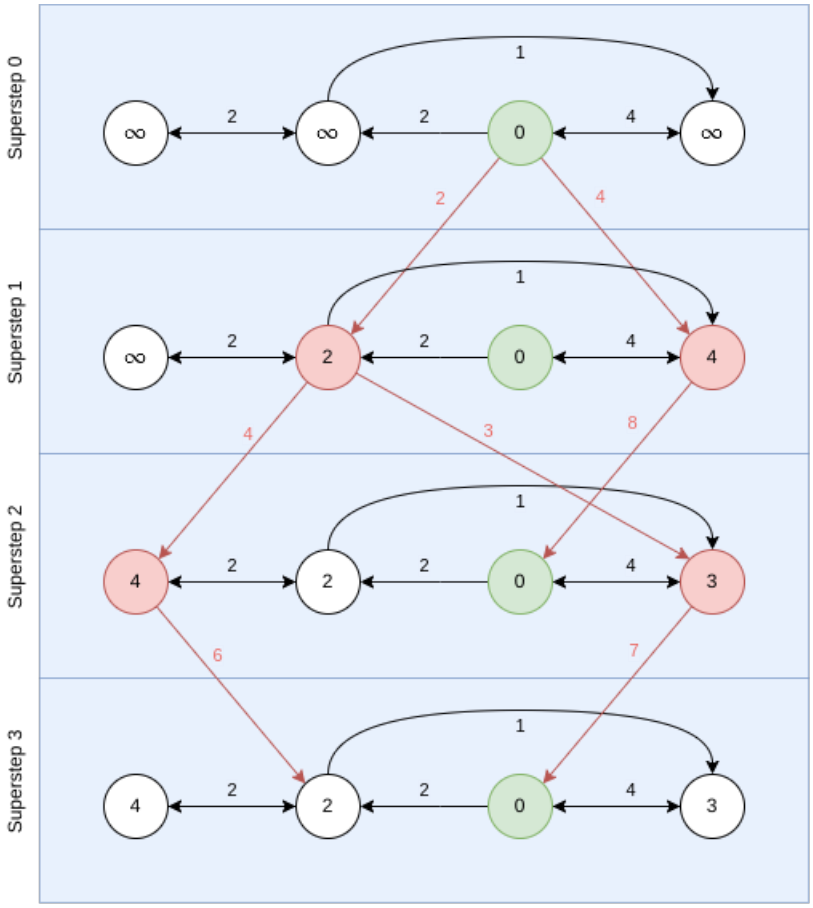


Example 5.2. Single-Source Shortest-Path

The following code computes the shortest path in a graph using the TLAV framework:

```
input:
  a graph G=(V,E)
  a starting vertex U

foreach v in V:
  shortest_path_L[v] = inf

send(0,U)

repeat
  for v in V do in parallel:
    minIncoming = min(receive(path_length));

    if minIncoming < shortest_path_L[v]:
      shortest_path_L[v] = minIncoming

      foreach e in E:
        path_length = shortest_path_L[v] + length[e]

        j = destination(e)
        send(path_length,j)
      end
    end
    halt() # if no message sent, then halt
  end
until no more messages are sent;
```

The execution could be as follows:

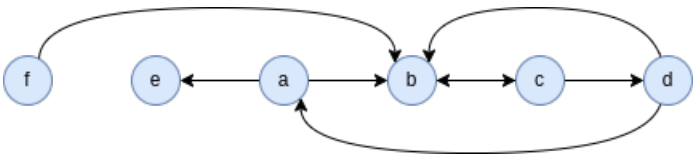


The colors indicate the same as in the previous example. The green node is the source node for the algorithm and the numbers indicate the weights of the edges.

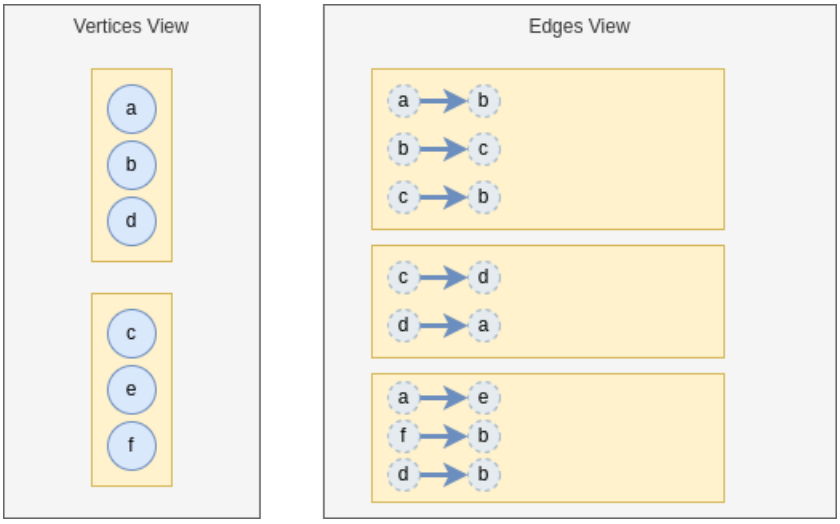
5.2.2 TLAV: Graph Distribution

We have seen that TLAV graph processing requires the graph data to be exposed in the form of two views: the set of vertices and the set of edges. Let's see how TLAV can be achieved in a distributed environment. Let's begin with an example:

Example 5.3. Consider the following distributed graph:

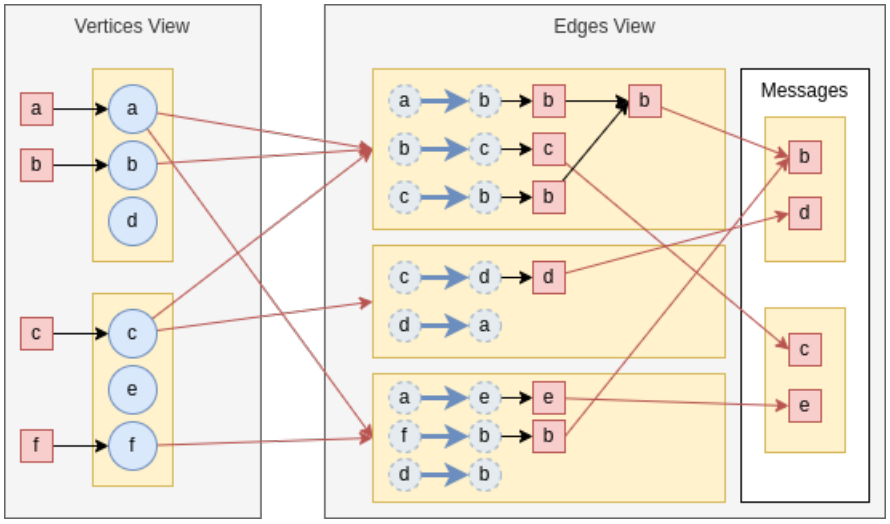


This graph can be represented as follows:



Here, we have depicted the two views, partitioned. Each partition is depicted by a yellow rectangle. And the instances are stored in the partitions. Note that the vertices do not store the nodes, I have draw them like that to facilitate readability.

Now, we want to set up a simple TLAV environment. We are going to send messages $\{a_m, b_m, c_m, f_m\}$. Each message is sent to the node it is named after. For example, a_m goes to node a. The nodes will send the message to all nodes to which they are connected, and these will generate a message named after them. For example, if b receive a message, it will send b_m. The first superstep of this process is:



Note how messages can be merged in two ways: intrapartition and interpartition. The messages in the white box would be sent to the vertices for the next superstep.

It is very important to realize that states can only be shared through messages, since there is no shared memory.

About the vertex kernel, keep in mind the after its execution, the vertices send messages to adjacent vertices, and these messages could be as complex as needed, and they can even modify the graph. The kernel must also include a **halt condition**, so that a node knows when to stop sending messages, and convergence can be achieved.

Pregel

Pregel is a very famous TLAV framework. Its model of computation is as follows:

- Input: a directed graph with three views:

  - List of nodes, uniquely identified.
  - List of edges.
  - Triplet views: an edge and its nodes information, showing all the information for the edge.

- Processing: in each superstep the vertices work in parallel.
  - They can modify their state, the state of their outgoing edges, receive messages from the previous superstep, send messages to other vertices for the next superstep and modify the graph topology.
  - The algorithm finishes based on a voting to halt. This means that when a node halts, it becomes inactive and stop sending processing and sending messages, until it receives another messages. When all nodes are inactive, the process finishes.

- Output: the set of values explicitly outputed by the vertices.

GraphX

GraphX is a subproject of Apache Spark, built as a Spark module and following Pregel's principles, but it only allows to send messahes to adjacent vertices.

It uses Spark GraphFrames to provide Pregel's required views (vertices, edges and triplets) and provides a library with typical distributed algorithms, such as pageRank, connected componentes, triangle count,...

5.2.3 Deepening into TLAV

Scheduling

Scheduling refers to how user-defined vertex programmes are scheduled for execution. They can be Synchronous, Asynchronous, Both or Hybrid.

Synchronous scheduling is based on the Bulk Synchronous Parallel (BSP) processing model. Active vertices are executed conceptually in parallel over one or more iterations, called supersteps and synchronization is achieved through a global synchronization barrier situated between each superstep that block vertices from computing the next superstep until all workers complete the current one. Each worker has to coordinate with the master to progress to the next superstep.

Synchronization is achieved because the barrier ensures that each vertex within a superstep has access to only the data from the previous superstep. Note that inside a superstep, vertices can be scheduled in a fixed or random order, because the execution order does not affect the state of the program (it should not, at least).

Pros os synchronous scheduling:

- Conceptually simple.
- Good for certain algorithms.
- Almost always deterministic, making synchronous applications easy to design, program, test, debug and deploy.
- Scalable: potentially linear in the number of vertices and can benefit from batch messaging between supersteps.

Cons:

- System throughput must remain high in each sueprstep ot justify the synchronization cost. Throughput is affected by the drop of active vertices or by the imbalance workload among workers, resulting in the system becoming underutilized; the iterative nature of graph algorithms, which suffer from 'the curse of the last reducer' (straggler problem), where many computations finish quickly but a small fraction of computations take a desproportionately larger amount of time; and the speed of computation of each node.
- The algorithm may not converge for some graph topologies. In general, algorithms that require some type of neighbor coordination may not always converge with the synchronous scheduling model without the use of some extra logic in the vertex program.

Asynchronous scheduling is different. There is no explicit synchronization points, so any active vertex is eligible for computation whenever processor and network resources are available. The vertex execution order can be dynamically generated and reorganized by the scheduler, and the straggler problem is eliminated. As a result, many asynchronous models outperform corresponding synchronous models, but at the expense of added complexity.

Pros:

- Outperform synchronous systems when the workload is imbalanced.

Cons:

- It cannot take advantage of batch messaging optimizations.
- The typical pull model execution may result in unnecessary processing.
- Asynchronous algorithms face more difficult scheduling problems and also consistency issues.

In general, synchronous execution generally accommodates IO bound algorithms, while asynchronous execution well-serves CPU bound algorithms by adapting to large and variable workloads.

Message Passing

Information is sent from one vertex program kernel to another via messages, which contain local vertex data and is addressed to the ID of the recipient vertex. A message can be addressed anywhere, but since vertices do not have ID information of all the other vertices, destination vertex IDs are typically obtained by iterating over outgoing edges.

After computation is complete and a destination ID for each message is determined, the vertex dispatches messages to the local worker process, which determines whether the recipient resides on the local machine or a remote one.

- If it is in the local machine, the worker process the message directly into the vertex's incoming messages queue.
- Otherwise, the worker process looks up the worker ID of the destination vertex and places the message in an outgoing message buffer, which are flushed when they reach a certain capacity, sending messages over the network in batches. In principle, it tries to wait until the end of a superstep to send all messages in batch-mode.

There three main strategies to optimize message passing:

1. **Sender-side combiner**: messages from several nodes are merged in the sender worker, which sends them to the destination worker.
2. **Receiver-side combiner**: in this case, the sender worker sends all the messages produced by all nodes to the destination worker, which makes the merging.
3. **Receiver-side scatter**: the sender worker send a message, which is received by the destination worker, sending it to several nodes.

Shared Memory

Shared memory exposes vertex data as shared variables that can be directly read or modigied by other vertex programs, avoiding the additional memory overhead constituted by messages. This is typical of centralized graph processing, but there are also some distributed systems that apply it.

The **main problem** is that for shared-memory TLAV frameworks, race conditions may arise when an adjacent vertex resides on a remote mahcine. Shared-memory TLAV frameworks often ensure memory consistency through mutual exclusion by requiring serializable schedules. **Serializability** means that every parallel execution has a corresponding sequential execution that maintains consistency.

The most prominent solutions up to today are:

- In GraphLab, border vertices are provided locally cached ghost copies of remote neighbors, where consistency between ghosts and the original vertex is maintained using pipelined distributed locking.

- In PowerGraph and GiraphX, graphs are partitioned by edges and cut along vertices, where consistency across cached mirrors of the cut vertex is maintained using parallel Chandy-Misra locking.

The reduced overhead of shared memory compared to message passing may lead to 35% faster converges when computing PageRank on a large web graph.

Partitioning

Large-scake graphs must be divided into parts to be placed in distributed storage/memory. Good partitions often lead to improved performance, but expensive strategies to partition can end up dominating processing time, leading many implementations to incorporate simple strategies, such as random placement.

Effective partitioning evenly distributes the vertices for balanced workload while minimizing imterpartition edges to avoid costly network traffic. This is formally known as a *k-way graph partitioning*, which is a NP-complete problem, with no fixed-factor approximation.

The leading work in graph partitioning can be broadly characterized as rigorous but impractical mathematical strategies or pragmatic heuristics used in practices, but this is currently an open problem.

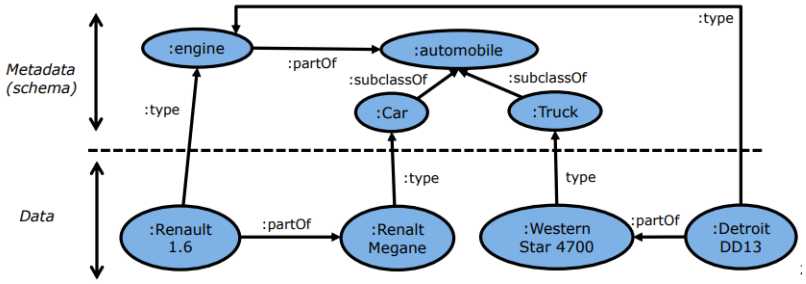


Figure 1: Example knowledge graph from [1].

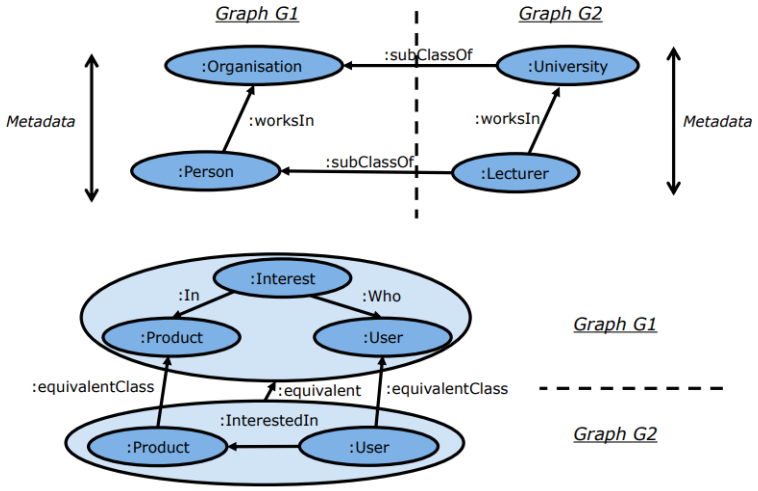


Figure 2: Examples of data linkage from [1].

Part II
Knowledge Graphs

6 Introduction to Knowledge Graphs

In a knowledge graph, every node is represented with a unique identifies and can be universally reffered, i.e., they can be referred potentially from any other database in the world.

Metadata is represented as nodes and edges in the graph.

Knowledge graphs facilitate linking data, because linking via their metadata is much more powerful than by the characteristics of the instances, and it is a unique feature of their own. In Figure 2, we can see several relationships between metadata nodes:

- A subClassOf B indicates that concept A is a subset (more specific) of concept B.
- A equivalentClass B indicates that classes A and B represent the same information, even if they are stored in different machines.
- A equivalent B also indicate equivalence of representation, but in this case it refers to subgraphs of the metadata.

Example 6.1. Assume Knowledge Graph as the canonical data model. First, model as graphs each source (separately):

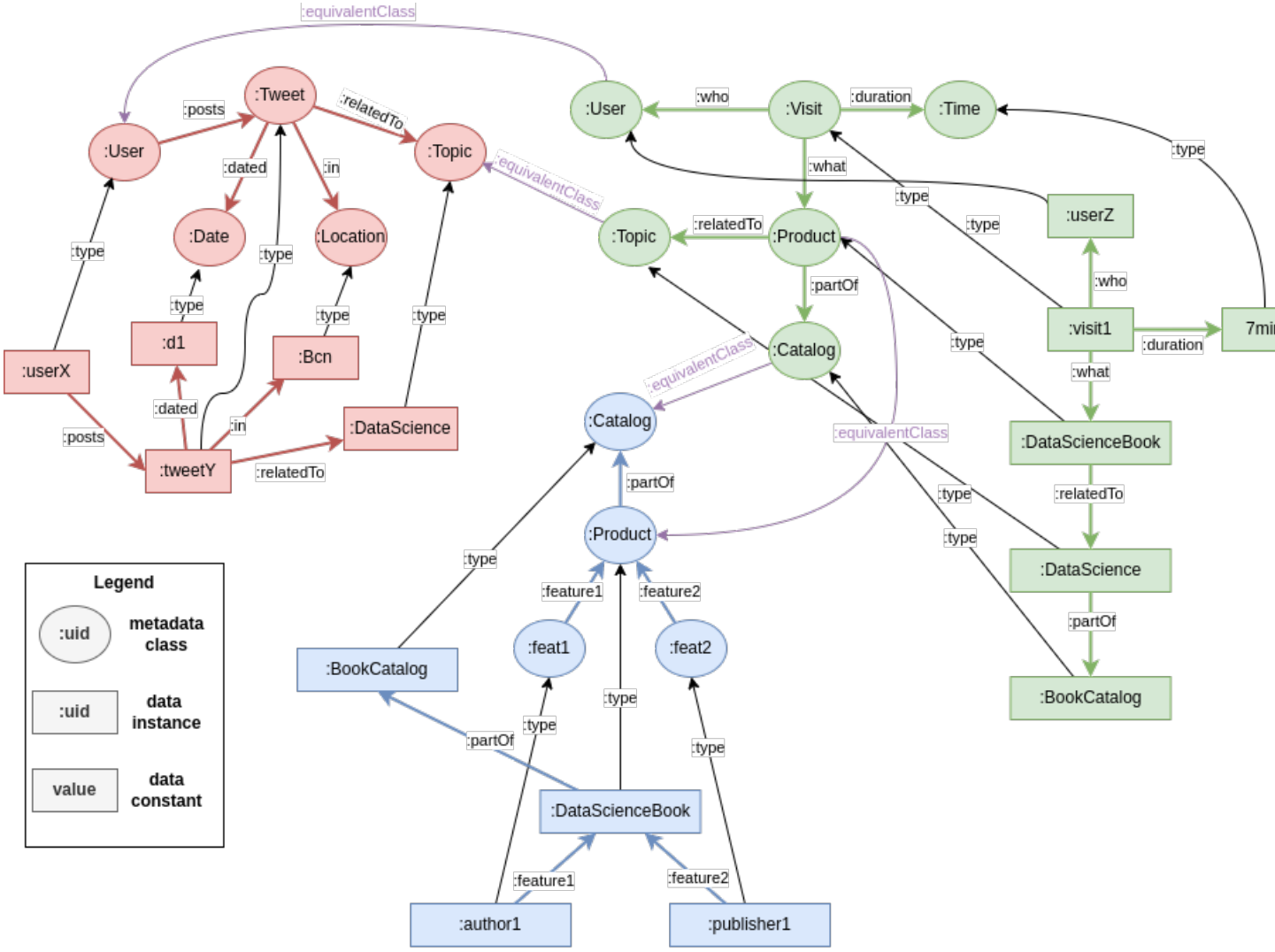
1. Model schema and some instances for each source:

Source 1	Source 2	Source 3
User		User
Tweet	Product	Product
Date	Product Features	Time
Location		

2. Then, relate the metadata from each graph with new edges generating a unique connected graph. For this:
   (a) Look for similar or identical concepts.
   (b) Think of interesting relationships you could exploit later.

Assume you can use the following pre-defined edges: equivalentClass, type and subClassOf, which embed the semantics already discussed.

A possible solution is the following:



Note that to model the ternary relationship between (User, Product, Time) we needed to add an artificial node. This is called **reification**.

Schema.org

Schema.org is a global initiative to mark up data. It provides a vocabulary of terms and their relationships. Google and others ahve built their semantic-aware searchers based on schema.org and built huge knowledge graphs based on it.

7 Resource Description Format (RDF)

RDF is a simple language for describing annotations (facts) about resources. It is the most basic ontology language. The triples that it uses as basic construct map to first order logic as grounded atomic formulas, and blank nodes map to existential variables.

The basic RDF block is the **RDF statement** which is a triple representing a binary relationship between two resources or between a resource and a literal. The syntax is:

<subject predicate object>

where:

- subject S has value object O for predicate P.
- subject and predicate are resources and must be URIs.
- object can be a resource (URI) or a literal (constant value).

As can be inferred from this, resources are identified by URIs, which are global identifiers. A URI is composed of a URL and a URN:

- URN is the Universal Resource Name: id
- URL is the Universal Resource Location: where it is

Many times, we omit the URL for simplicity, and refer to the URI as :URN.

A **blank node** is a resource without a URL (i.e., _).

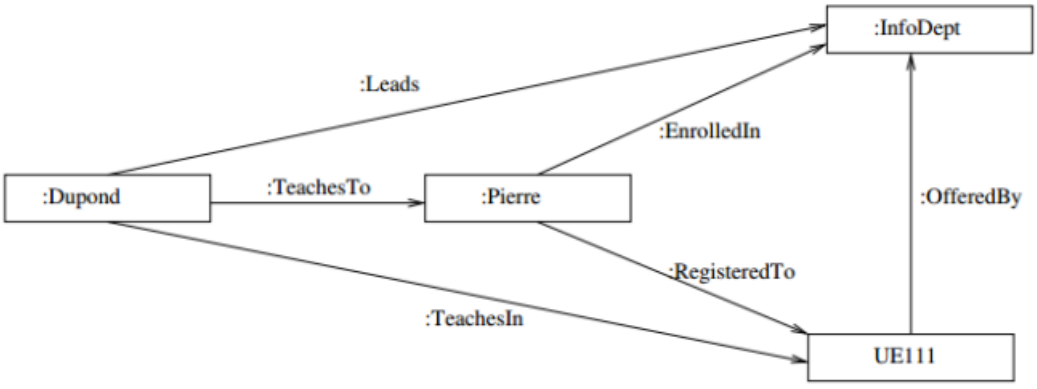
Literals are atomic values such as strings, dates or numbers.

We can thus define a **RDF graph** (or semantic graph) as a set of these RDS statements.

To query RDF graph, SPARQL is the de facto language (also for it variants and extensiosn). It is inspired by SQL but oriented to express graph operations.

Example 7.1. A RDF graph example.

The graph:



Can be represented as an RDF like this:

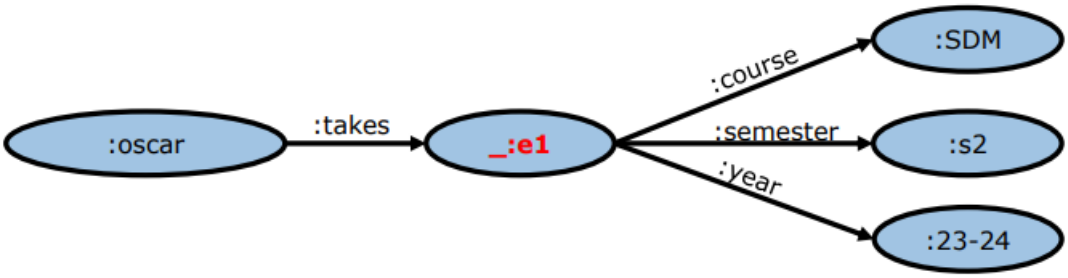
```
(:Dupond :Leads :CSDept)
(:Dupond :TeachedIn :UE111)
(:Dupond :TeachesTo :Pierre)
(:Pierre :EnrolledIn :CSDept)
(:Pierre :RegisteredTo :UE111)
(:UE111 :OfferedBy :CSDept)
```

Usually, RDF is serialized using the XML format.
The rdf URL is a namespace for RDF.
Other RDF syntaxes are turtle (which is human-readbale), N-triples or Notation 3.

7.1 RDF Modeling

RDF modeling is based on binary relationships, but n-ary relationships may be needed, so blank nodes were presented as a solution for this. A **blank node** is a node without a URI, which cannot be referenced and can only be subjects or objects. Its semantics are not completely clear yet, but their **de facto use** is as an identifier without a URI. The W3C position is this regard is to use blank nodes for incomplete data: unknown values or anonymized values. The de facto use is pragmatic, but good practices discourage their use: all resources should have a proper URI.

Example 7.2. Quoting
The following use of a blank node:



Can be expressed as:
:oscar :takes [:course :SDM]
This is quoting, which is general is [property object], and the subject is the blank node.

Notice that we cannot express neither schema nor additional constraints, such as 'at least one' or 'at most three'.

7.2 RDF-star

RDF-star in as RDF extension, more comapct and with a predice syntax for reification.

Example 7.3. The following is RDF-star:

```
@prefix: <http://www.example.org/>

:employee38 :familyName "Smith"
:employee22 :claims << :employee38 :jobTitle "Assistant Designer" >>
```

Here, we have what is called an **embedded triple**, and it models the 3-way relationship between (:emp22, :emp38, "Assistant Designer").

SPARQL-star is an extension of SPARQL to query RDF-star graphs.

8 RDF Schema (RDFS)

RDFS extends RDF to not only consider data instances, but also schema. In this case, we can define classes and relationships between them, using the same principles as for instances. It defines *rdfs:*, a namespace for RDFS, in which a set of resources needed to express contraints is defined.
RDFS allows to specify the following constraints:

- Declare resources as instances of certain classes.
  - :Oscar :type :lecturer

- Inclusion statements between classes and between properties, which allow us to define **taxonomies**.
  - :lecturer :subclassOf :human
- Assert the class of a subject (or an object) of a property
  - The predicate :parentOf must relate instnaces of the class :human. Therefore, any subject/object of a triple where :parentOf is the predicate is atumatically asserted as :human.

8.1 RDFS statements at the schema level

- Instances:
  - :Oscar rdf:type :Lecturer
- Taxonomies:
  - Classes: :Lecturer rdfs:subclassOf :Human
  - Relationships: :ResponsibleFor rdfs:subpropertyOf :Lecutres
- Domain and Range:
  - :Lectures rdfs:domain :Human
  - :Lectures rdfs:range :Course

8.2 RDFS Core Classes

- **rdfs:Resource**: the class of all resources. Everything is a resource.
- **rdfs:Class**: the class of all classes.
- **rdfs:Literal**: the class of all literals.
- **rdf:Property**: the class of all properties.
- **rdf:Statement**: the class of all statements.

These core classes add another level of abstraction above the metadata layer.

8.3 RDFS Inference

In RDFS we can infer new instances (knowledge) from the statements created by the users. It is based on **rule-based reasoning** and there are two kinds of inference:

- **Core type inference**: it infers the type with regards to the core classes of an asserted resource R.
- **Domain-specific inference**: it can be:
  - Inclusion dependencies:
    - If :A rdfs:subclassOf :B and :B rdfs:subclassOf :C then :A rdfs:subclassOf :C.
  - Type inference: it infers the type of an asserted resource R with respect to a user created class X.

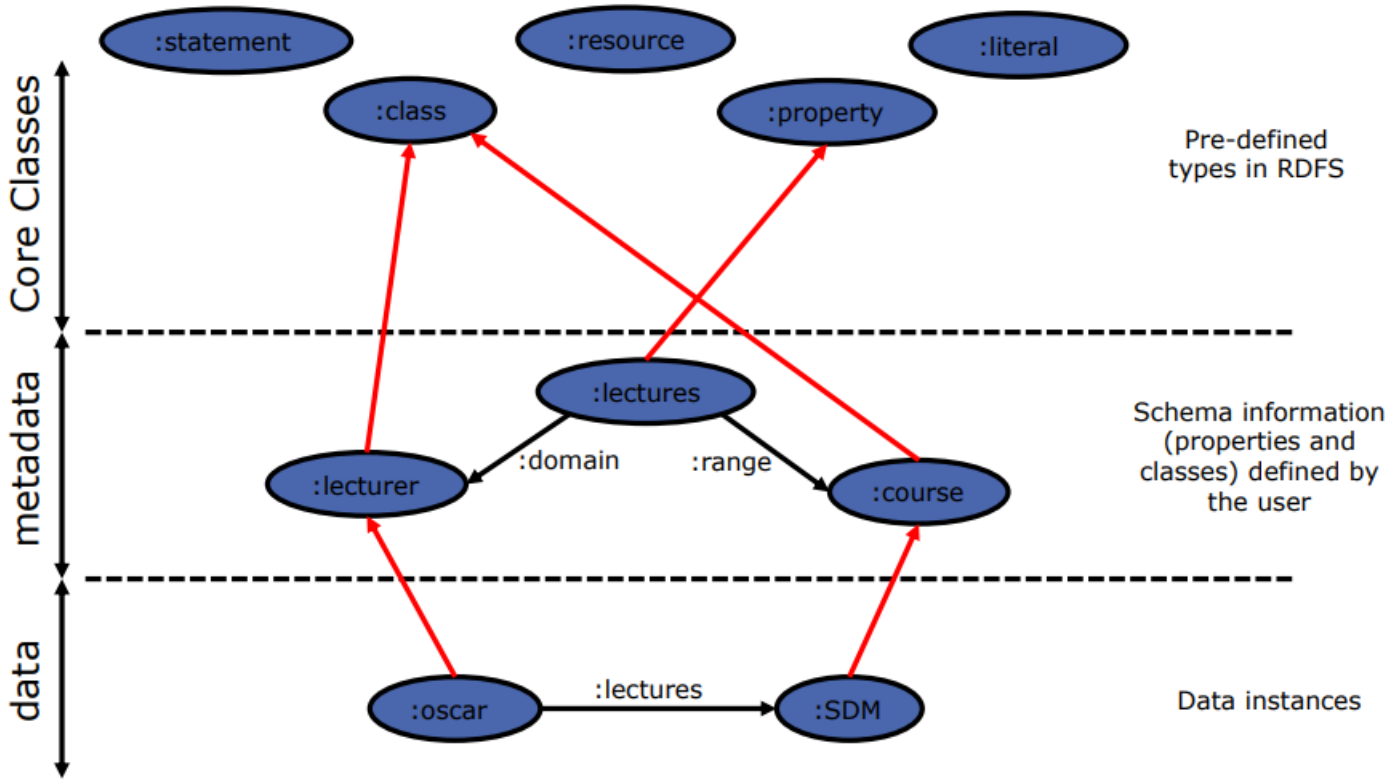


Figure 3: Three layers of abtraction of RDFS from [1]. The red arrows are *rdf:type* relationships.

8.4 RDFS Core Properties

- **rdf:type**: relates a resource to its class:

  :oscar rdf:type :lecturer

  The subject resource (:oscar) is declared to be an instance of the object class (:lecturer).

  - The inferred knowledge is:
    - Core type inference: the object is inferred as a Class:

      :lecturer rdf:type rdfs:Class

- **rdfs:subClassOf**: relates a class to one of its superclasses:

  :lecturer rdfs:subClassOf :human

  :oscar rdf:type :lecturer

  The subject and object are declared as classes and any instance of the subject is declared as an instance of the object.

  - The inferred knowledge is:
    - Core type inference: the subject and object are inferred as classes:

      :lecturer rdf:type rdfs:Class

      :human rdf:type rdfs:Class
    - Domain specific inference: inclusion dependency:

      :oscar rdf:type :human

- **rdfs:subPropertyOf**: relates a property to one of its superproperties:

  :responsibleFor rdfs:subPropertyOf :lectures

  :oscar :responsibleFor :OD

  The subject and object resources are declared to be properties and any subject, object related by subject predicate are automatically declared as to be related by the object predicate.

  - The inferred knowledge is:
    - Core type inference: the subject and object are inferred as properties:

      :responsibleFor rdf:type rdf:Property

      :lectures rdf:type rdf:Property
    - Domain specific inference: inclusion dependency:

      :oscar :lectures :OD

- **rdfs:domain**: specifies the domain of a property:

  :lectures rdf:domain :lecturer

  :oscar :lectures :OD

  - The inferred knowledge is:
    - Core type inference: the subject is declared to be a property and the object is declared to be a class:

      :lectures rdf:type rdf:Property

      :lecturer rdf:type rdfs:Class
    - Domain specific inference: type inference:

      :oscar rdf:type :lecturer

- **rdfs:range**: specifies the range of a property:

  :lectures rdf:range :course

  :oscar :lectures :OD

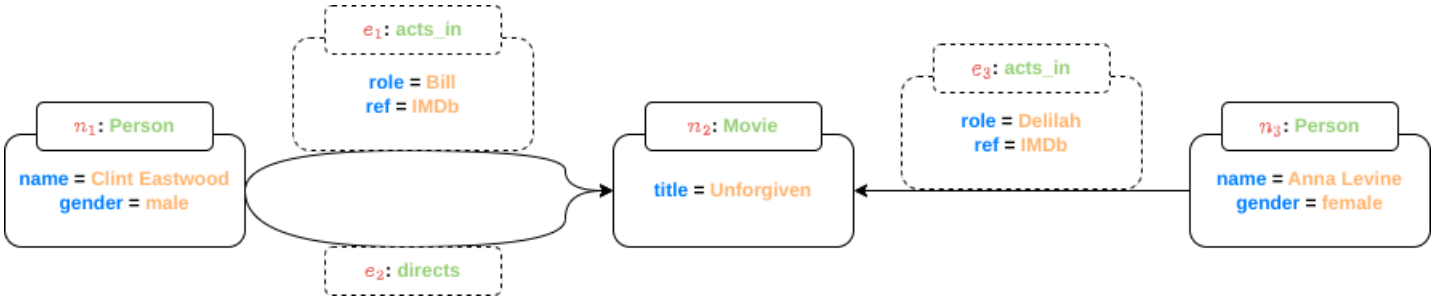
  - The inferred knowledge is:
    - Core type inference: the subject is declared to be a proeprty and the object is declared to be a class:

      :lectures rdf:type rdf:Property

      :course rdf:type rdfs:Class
    - Domain specific inference: type inference:

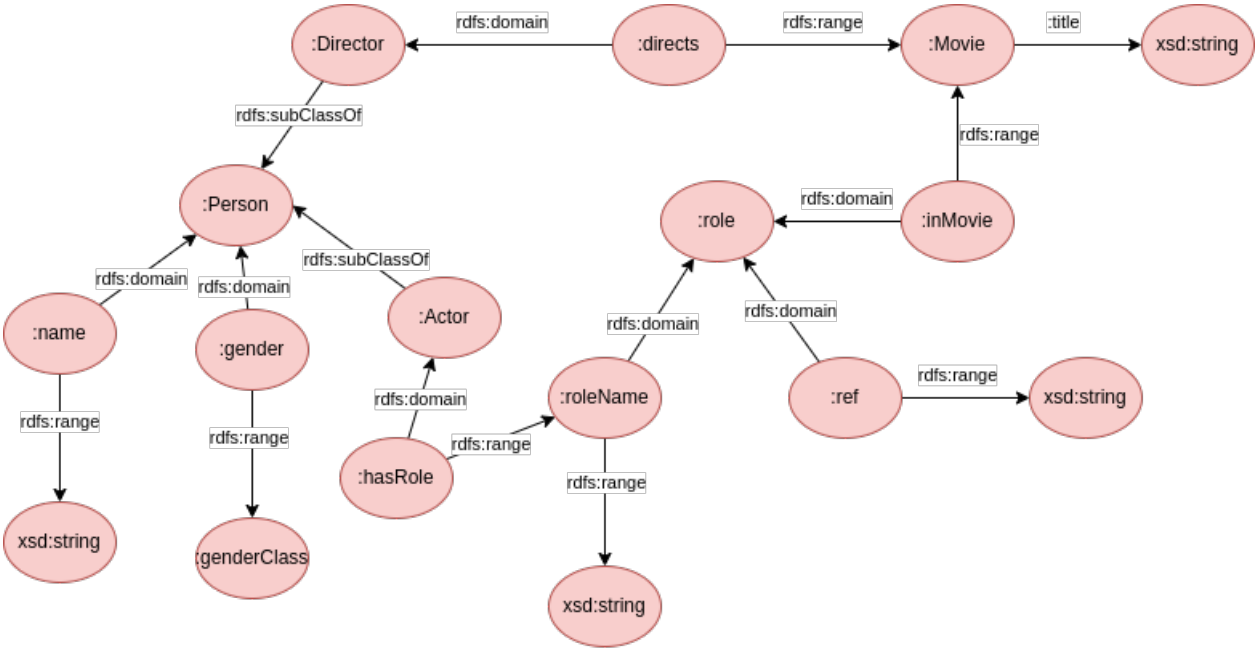
      :OD rdf:type :course

Example 8.1. An RDFS graph.
Consider the graph that we have used many times already:



Create a correct RDFS graph capturing as much constraints as possible from it. What triples may you infer from the asserted RDFS graph?

The RDFS graph can be the following:



And the instantiation of the data:

Instantiation	Inference
:clintEastwood :hasRole :bill	:clintEastwood rdf:type Actor :bill rdf:type :role :clintEastwood rdf:type Person
:bill :inMovie :Unforgiven	:bill rdf:type :role :Unforgiven rdf:type :Movie
:Ungorgiven :title 'Unforgiven'	:Unforgiven rdf:type :Movie
:clintEastwood :name 'Clint Eastwood'	:clintEastwood rdf:type :Person
:clintEastwood :gender :male	:clintEastwood rdf:type :Person :male rdf:type :genderClass
:bill :roleName 'Bill'	:bill rdf:type :role
:bill :red 'IMDB'	:bill rdf:type :role
:annaLevine :hasRole :delilah	:annaLevine rdf:type :Actor :delilah rdf:type :role :annaLevine rdf:type :Person
:delilah :inMovie :Unforgiven	:delilah rdf:type :role :Unforgiven rdf:type :Movie
:annaLevine :name 'Anna Levine'	:annaLevine rdf:type :Person
:annaLevine :gender :female	:annaLevine rdf:type :Person :female rdf:type :genderClass
:delilah :roleName 'Delilah'	:delilah rdf:type :role
:delilah :ref 'IMDB'	:delilah rdf:type :role
:clintEastwood :directs :Unforgiven	:clintEastwood rdf:type :Director :Unforgiven rdf:type :Movie :clintEastwood rdf:type Person

8.5 SPARQL

SPARQL Protocol And RDF Query Language (SPARQL) is the standard query language for RDF(S) graphs, being also a W3C recommendation and supporting RDFS and OWL under specific entailments.

SPARQL is based on navigational pattern matching, and simple RDF graphs are used as query patterns. The semantics applied are homomorphism semantics.

Example 8.2. A simple query:

```
SELECT x, z
WHERE
  x Lectures y,
  y TaughtIn z,
  z rdf:type Faculty
```

SPARQL has 4 basic forms that retrive either result sets or RDF graphs:

- SELECT: returns all, or a subset of, the variables bound in a query pattern match.
- CONSTRUCT: returns an RDF graph constructed by substituting variables in a set of triple templates.
- ASK: returns a boolean indicating whether a query pattern is matched or not.
- DESCRIBE: returns an RDF graph that describes the resources found.

A **SPARQL Endpoint** is an endpoint accepting SPARQL queries and returning the results via HTTP.

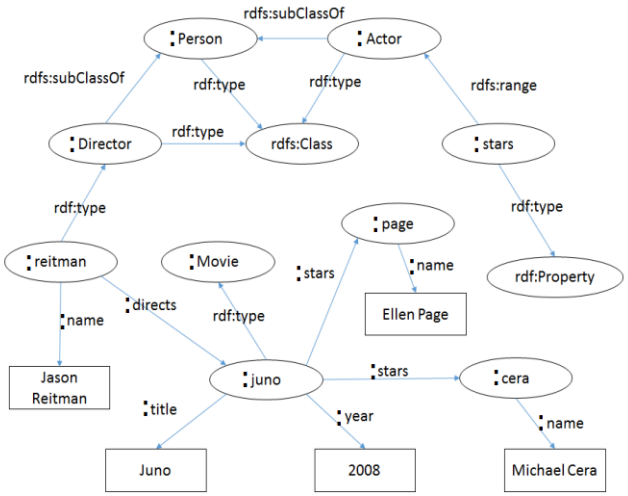
Example 8.3. A more complete example:

```
PREFIX fib: <http://www.fib.edu/elements/>
SELECT ?lecturer ?course
WHERE
  {
    ?lecturer fib:lectures ?course
  }
```

This query is equivalent to: SELECT x,y WHERE x LECTURES y.

SPARQL allows property paths based on regular expressions.

Example 8.4. Take into account the following graph:



And write the following queries, assuming no entailment regime:

1. Get the name of all actors that participated in Juno:

```
PREFIX mov: <url>
SELECT ?name
WHERE
  {
    mov:juno mov:stars ?actor.
    ?actor mov:name ?name
  }
```

2. Get the name of all directors:

```
PREFIX mov <url>
SELECT ?name
WHERE
  {
    ?dir rdf:type :Director.
    ?dir mov:name ?name
  }
UNION
SELECT ?name
WHERE
  {
    ?dir mov:directs ?mov.
    ?dir mov:name ?name
  }
```

3. Get the name of all persons:

```
PREFIX mov: <url>
SELECT ?name
WHERE
  {
    ?person mov:name ?name
  }
```

4. Get the title of all movies:

```
PREFIX mov: <url>
SELECT ?title
WHERE
  {
    ?movie mov:title ?title.
    ?movie rdf:type ?movie
  }
```

8.5.1 Entailment Regimes

The most basic entailment regime supported by SPARQL is **simple entailment**, in which graph patterns are evaluated by means of pattern matching under homomorphism semantics.

Nonetheless, more elaborate entailment relations have been developed, to retrieve solutions that are logical consequences of the axioms asserted. The most popular ones are RDFS entailment and OWL 2 RDF-Based Semantics entailment.

8.6 RDFS Inference Rules

The RDFS entailment rules are presented in the following table:

	If S contains	Then S RDFS entails recognizing D:
rdfs1	any IRI a in D	a rdf:type rdfs:Datatype
rdfs2	a rdfs:domain x. y a z.	y rdf:type x
rdfs3	a rdfs:range x. y a z	z rdf:type x
rdfs4a	x a y	x rdf:type rdfs:Resource
rdfs4b	x a y	y rdf:type rdfs:Resource
rdfs5	x rdfs:subPropertyOf y. y rdfs:subPropertyOf z	x rdfs:subPropertyOf z
rdfs6	x rdf:type rdf:Property	x rdfs:subPropertyOf x
rdfs7	a rdfs:subPropertyOf b. x a y	x b y
rdfs8	x rdf:type rdfs:Class	x rdfs:subClassOf rdfs:Resource
rdfs9	x rdfs:subClassOf y. z rdf:type x	z rdf:type y
rdfs10	x rdf:type rdfs:Class	x rdfs:subClassOf x
rdfs11	x rdfs:subClassOf y. y rdfs:subClassOf z	x rdfs:subClassOf z
rdfs12	x rdf:type rdfs:ContainerMembershipProperty	x rdfs:subPropertyOf rdfs:Member
rdfs13	x rdf:type rdfs:Datatype	x rdfs:subClassOf rdfs:Literal

8.6.1 The RDFS Paradox

The Russel's Paradox is a theoretical paradox that arises within a naïve set theory, by considering the set of all sets that are not members of themselves. This paradox arises in RDFS. This means that the RDFS regime entailment is flawed, and the reason is that the RDFS core classes and properties are ill-defined. The problems are:

- rdfs:Class is an instance of itself. This allows the possibility of having inifinitely many layers of classes.
- rdfs:Resource is a superclass and an instance of rdfs:Class at the same time.
- rdfs:subClassOf, rdf:type, rdfs:Range and rdfs:Domain are both used to define the other RDFS primitives and the user metadata.

The SPARQL community rethought the RDFS metamodel to introduce **fix-point reasoning**, disallowing for infinite inference loops, and having elements organized in a strict order, i.e., an element cannot be an element and a set at the same time, and no element can be placed twice in a taxonomy at different levels. This gave birth to the **modified RDFS Entailment Regime**:

	If S contains	Then S RDFS entails recognizing D:
rdfs1	any IRI a in D	a rdf:type rdfs:Datatype
rdfs2	a rdfs:domain x. y a z.	y rdf:type x
rdfs3	a rdfs:range x. y a z	z rdf:type x
rdfs4a	x a y	x rdf:type rdfs:Resource
rdfs4b	x a y	y rdf:type rdfs:Resource
rdfs5	x rdfs:subPropertyOf y. y rdfs:subPropertyOf z	x rdfs:subPropertyOf z
rdfs6	x rdf:type rdf:Property	x rdfs:subPropertyOf x
rdfs7	a rdfs:subPropertyOf b. x a y	x b y
rdfs8	x rdf:type rdfs:Class	x rdfs:subClassOf rdfs:Resource
rdfs9	x rdfs:subClassOf y. z rdf:type x	z rdf:type y
rdfs10	x rdf:type rdfs:Class	x rdfs:subClassOf x
rdfs11	x rdfs:subClassOf y. y rdfs:subClassOf z	x rdfs:subClassOf z
rdfs12	x rdf:type rdfs:ContainerMembershipProperty	x rdfs:subPropertyOf rdfs:Member
rdfs13	x rdf:type rdfs:Datatype	x rdfs:subClassOf rdfs:Literal

9 Ontology Languages: Description Logics

Definition 9.1. A **ontology** is a formal description of a domain in terms of the concepts and roles or properties between them. More precisely, it is a controlled vocabulary or schema, usually called the **TBOX**, with aligned instances, or **ABOX**.
The TBOX and ABOX assertions are described with formal semantics and, based on its formal semantics, it defines inference rules, based on some kind of reasoning.

Note that RDF graphs are not ontologies and that RDFS are ontologies only if we take care of making them to be, by following the good practices. OWL graphs (we will see them later) are forced to be ontologies.

9.1 Logic Based Ontology Languages

First Order Logic (FOL) is suitable for knowledge representation, since classes can be represented as unary predicates, properties/relationship as binary predicates and constraints as logical formulas using the predicates.

Nonetheless, we must take into consideration of undecidability problem: there is no algorithm that determines if a FOL formula implies another. Therefore, we have to work with decidable fragments of FOL:

- **Description logics**: binary predicates with bounded number of variables.
- **Datalog**: Horn-clauses.

The characteristics of these are summarized in the following table:

	Datalog	Description Logics
Focus	Instances	Knowledge
Approach	Centralized	Decentralized
Reasoning	Closed-World Assumption	Open-World Assumption
Unique name	Unique name assumption	Non-unique name assumption

The open-world assumption implies that everything can be true, unless the opposite is explicitly indicated.

9.2 TBOX

A TBOX is characterized by a set of constructs for building complex concepts and roles from atomic ones.

- **Concepts** correspond to classes.
- **Roles** correspond to relationships.

Then, the TBOX defines the terminology of the domain, with formal semantics given in terms of interpretations:

Definition 9.2. An **interpretation** $\mathcal{I} = \left(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}\right)$ consists of a nonempty set $\Delta^{\mathcal{I}}$, the **domain** of $\mathcal{I}$, and an **interpretation function** $\cdot^{\mathcal{I}}$, which maps:

- Each individual, i.e., each element in the real world that we want to represent, a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.
- Each atomic concept A to a subset $A^{\mathcal{I}} \subset \Delta^{\mathcal{I}}$.
- Each atomic role P to a subset $P^{\mathcal{I}} \subset \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

With these basic pieces, we can construct more complex concepts:

Construct	Syntax	Semantics	Example
atomic concept	A	$A^{\mathcal{I}} \subset \Delta^{\mathcal{I}}$	Doctor
atomic role	P	$P^{\mathcal{I}} \subset \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	hasChild
atomic negation	$\neg A$	$\Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$	¬Doctor
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	Human⊓Male
unqualified existence restriction	$\exists R$	$\left\{a \mid \exists b : (a,b) \in R^{\mathcal{I}}\right\}$	∃hasChild
value restriction	$\forall R.C$	$\left\{a \mid \forall b, (a,b) \in R^{\mathcal{I}} \implies b \in C^{\mathcal{I}}\right\}$	∀hasChild.Male
bottom	$\bot$	$\emptyset$	

In this table, C and D denote arbitrary concepts, i.e., a combination of atomic concepts through appropriate constructs, and R an arbitrary role, i.e., a combination of atomic roles through appropriate constructs.

The combination of these constructrs form the basic language $\mathcal{AL}$ of the family of $\mathcal{AL}$ languages. However, this can be extended, adding new constructs:

Construct	$\mathcal{AL}\cdot$	Syntax	Semantics	Example
disjunction	$\mathcal{U}$	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	$(Human \sqcap Doctor) \sqcup (Human$
top		$\top$	$\Delta^{\mathcal{I}}$	
qualified existence restriction	$\mathcal{E}$	$\exists R.C$	$\left\{a \mid \exists b : (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\right\}$	∃Treats.Doctor
full negation	$\mathcal{C}$	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	$\neg\left(\exists Treats.Docto$
number restrictions	$\mathcal{N}$	$(\geq k\ R)$	$\left\{a \mid \#\left\{b : (a,b) \in R^{\mathcal{I}}\right\} \geq k\right\}$	≥ 5 Treats
		$(\leq k\ R)$	$\left\{a \mid \#\left\{b : (a,b) \in R^{\mathcal{I}}\right\} \leq k\right\}$	≤ 5 Treats
qualified number restriction	$\mathcal{Q}$	$(\geq k\ R.C)$	$\left\{a \mid \#\left\{b : (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\right\} \geq k\right\}$	≥ 5 Treats.$(Doctor \sqcap$
		$(\leq k\ R.C)$	$\left\{a \mid \#\left\{b : (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\right\} \leq k\right\}$	≤ 5 Treats.$(Doctor \sqcap F$
inverse role	$\mathcal{I}$	R^-	$\left\{(b,a) \mid (a,b) \in R^{\mathcal{I}}\right\}$	Treats$^-$
role closure	reg	R^*	$\left(R^{\mathcal{I}}\right)^*$	$(Treats)^*$

Example 9.1. What is the meaning of these axioms:

- Disjunction:

$$\forall hasChild.\,(Doctor \sqcup Lawyer)$$

Formally:

$$\left\{a \mid \forall b : (a,b) \in hasChild^{\mathcal{I}} \implies b \in Doctor^{\mathcal{I}} \vee b \in Lawyer^{\mathcal{I}}\right\}$$

Natural language: all entities such that all their childs are a doctor or a lawyer.

- Qualified existencial restriction:

$$\exists hasChild.Doctor$$

  Formally:

$$\left\{a | \exists b : (a,b) \in hasChild^{\mathcal{I}} \wedge b \in Doctor^{\mathcal{I}}\right\}$$

  Natural language: all entities that has a child who is a doctor.

- Full negation:

$$\neg \left(Doctor \sqcup Lawyer\right)$$

  Formally:

$$\Delta^{\mathcal{I}} \setminus \left(Doctor^{\mathcal{I}} \cup Lawyer^{\mathcal{I}}\right)$$

  Natural language: all entities that not a doctor nor a lawyer.

- Number restrictions:

$$(\geq 2\ hasChild) \sqcap (\leq 1\ sibling)$$

  Formally:

$$\left\{a | \#\left\{b : (a,b) \in hasChild^{\mathcal{I}}\right\} \geq 2 \wedge \#\left\{c : (a,c) \in siblings^{\mathcal{I}}\right\} \leq 1\right\}$$

  Natural language: all entities that have at least two childs and at most one sibling.

- Qualified number restrictions:

$$(\geq 2\ hasChild.Doctor)$$

  Formally:

$$\left\{a | \#\left\{b : (a,b) \in hasChild^{\mathcal{I}} \wedge b \in Doctor^{I}\right\} \geq 2\right\}$$

  Natural language: all entities that have at least two childs that are doctors.

- Inverse role:

$$\forall hasChild^{-}.Doctor$$

  Formally:

$$\left\{a | \forall b : (b,a) \in hasChild^{\mathcal{I}} \implies b \in Doctor^{\mathcal{I}}\right\}$$

  Natural language: all entities that are childs of doctors.

- Reflexive-transitive role closure:

$$\exists hasChild.Doctor$$

  Formally:

$$\left\{a | \exists b : (a,b) \in \left(hasChild^{\mathcal{I}}\right)^{*} \wedge b \in Doctor^{\mathcal{I}}\right\}$$

  Natural language: all entities that are ascendent of some doctor.

A Description Logics TBOX only includes terminological axioms of the following form:

1. **Inclusion**:
   (a) $C_1 \sqsubseteq C_2$ is satisfied by $\mathcal{I}$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$.
   (b) $R_1 \sqsubseteq R_2$ is satisfied by $\mathcal{I}$ if $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$.
2. **Equivalence**:
   (a) $C_1 \sqsubseteq C_2, C_2 \sqsubseteq C_1$.

Example 9.2. A TBOX example
The following axioms define a TBOX:

$$\begin{aligned} Woman &\equiv Person \sqcap Female \\ Man &\equiv Person \sqcap \neg Woman \\ Mother &\equiv Woman \sqcap \exists hasChild.Person \\ Father &\equiv Man \sqcap \exists hasChild.Person \\ Parent &\equiv Father \sqcup Mother \\ Grandmother &\equiv Mother \sqcap \exists hasChild.Parent \\ MotherWithManyChildren &\equiv Mother \sqcap \geq 3\ hasChild \\ MotherWithoutDaughter &\equiv Mother \sqcap \forall hasChild.\neg Woman \\ Wife &\equiv Woman \sqcap \exists hasHusband.Man \end{aligned}$$

9.3 ABOX

The ABOX defines the instances in terms of the terminological axioms defined in the TBOX, by using concept $(Student\,(Pere))$ and role $(Teaches\,(Oscar, Pere))$ assertions.

Example 9.3. A knowledge base (TBOX+ABOX)
The TBOX assertions are the following:

- Inclusion assertions on concepts:

$$\begin{aligned} Father &\equiv Human \sqcap Male \sqcap \exists hasChild \\ HappyFather &\sqsubseteq Father \sqcap \forall hasChild.\,(Doctor \sqcup Lawyer \sqcup HappyPerson) \\ HappyAnc &\sqsubseteq \forall descendant.HappyFather \\ Teacher &\sqsubseteq \neg Doctor \sqcap \neg Lawyer \end{aligned}$$

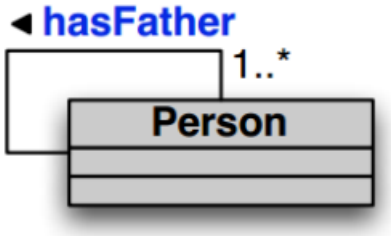
- Inclusion assertions on roles:

$$\begin{aligned} hasChild &\sqsubseteq descendant \\ hasFather &\sqsubseteq hasChild^{-} \end{aligned}$$

The ABOX membership assertions are:

$$\begin{aligned} &Teacher\,(mary) \\ &hasFather\,(mary, john) \\ &HappyAnc\,(john) \end{aligned}$$

Example 9.4. The following UML diagram



can be represented as the TBOX:

$$\begin{aligned} \exists hasFather &\sqsubseteq Person \\ \exists hasFather^{-} &\sqsubseteq Person \\ Person &\sqsubseteq \exists hasFather \end{aligned}$$

9.4 Models of a Description Logics Ontology

Definition 9.3. A **model** of a knowledge base $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ is an interpretation $\mathcal{I}$ that satisfies all assertions in $\mathcal{T}$ and all assertions in $\mathcal{A}$.
If there is a model for $\mathcal{O}$, then it is **satisfiable**.
$\mathcal{O}$ **logically implies** an assertion α, written $\mathcal{O} \models \alpha$ if α is satisfied by all models of $\mathcal{O}$.

Satisfiability looks for contrasictions in the asserted axioms. Without negations, everything is satisfiable, and TBOX axioms are just used to infer knowledge for the asserted elements. This is the case of RDFS, and in case of an error, the knowledge base will simply infer wrong knowledge, but no error nor alert will be rased.

Including negation, we can identify mistakes in the ABOX, since such interpretations will not be a model for that ontology.

Example 9.5. Is the following interpretation a model? Can you think of an interpretation that is a model?
The TBOX is:

$$\begin{aligned} \exists Teaches &\sqsubseteq Teacher \\ \exists Teaches^{-} &\sqsubseteq Course \\ Teacher &\sqsubseteq \neg Course \end{aligned}$$

And the ABOX is:

$$Teaches\,(x, x)\,.$$

It is not a model, because

$$Teaches\,(x, x) \overset{\exists Teaches \sqsubseteq Teacher}{\models} Teacher\,(x) \overset{Teacher \sqsubseteq \neg Course}{\models} \neg Course\,(x)$$

and

$$Teaches\,(x) \overset{\exists Teaches^{-} \sqsubseteq Course}{\models} Course\,(x)\,,$$

and both expressions cannot evaluate to true at the same time.

To make a model, we can change the ABOX to be

$$Teaches\,(x, y)\,,$$

in which case all assertions from the TBOX and the ABOX can be satisfied.

Example 9.6. Description Logics Reasoning
Consider the following TBOX and make all possible inferences:

$$\begin{aligned} Researcher &\sqsubseteq \neg Professor \\ Researcher &\sqsubseteq \neg Lecturer \\ \exists TeachesTo^{-} &\sqsubseteq Student \\ Student \sqcap \neg Undergrad &\sqsubseteq GraduateStudent \\ \exists TeachesTo.Undergrad &\sqsubseteq Professor \sqcup Lecturer \end{aligned}$$

The TBOX inferences are:

$$Researcher \sqsubseteq \exists TeachesTo.GraduateStudent$$

Do the same with this ABOX:

$$TeachesTo\,(dupond, pierre)$$
$$\neg GraduateStudent\,(pierre)$$
$$\neg Professor\,(dupond)$$

The ontology inferences are:

$$Undergrad\,(pierre)$$
$$Lecturer\,(dupond)$$

9.4.1 TBOX Reasoning

- **Concept Satisfiability**: a concept C is satisfiable with respect to the TBOX $\mathcal{T}$ if there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}}$ is not empty, i.e., $T \not\models C \equiv \bot$.
- **Subsumption**: a concept C_1 is subsumed by another concept C_2 with respect to the TBOX $\mathcal{T}$ if, for every model $\mathcal{I}$ of $\mathcal{T}$, we have $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$, i.e., $\mathcal{T} \models C_1 \sqsubseteq C_2$.
- **Equivalence**: C_1 and C_2 are equivalent with respect to $\mathcal{T}$ if, for every model $\mathcal{I}$ of $\mathcal{T}$ we have $C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$, i.e., $\mathcal{T} \models C_1 \equiv C_2$.
- **Disjointness**: C_1 and C_2 are disjoint with respect to $\mathcal{T}$ if, for every model $\mathcal{I}$ of $\mathcal{T}$, we have $C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} = \emptyset$, i.e., $\mathcal{T} \models C_1 \sqcap C_2 \equiv \bot$.
- **Functionality implication**: a functionality assertion, $(funct\ R)$, is logically implied by $\mathcal{T}$ if, for every model $\mathcal{I}$ of $\mathcal{T}$,we have that $(o, o_1) \in R^{\mathcal{I}}$ and $(o, o_2) \in R^{\mathcal{I}}$ implies $o_1 = o_2$, i.e., $\mathcal{T} \models (funct\ R)$.

9.4.2 Reasoning complexity

In the next table, the complexity for concept satisfiability for each of the logic families we have seen is shown:

Family	Complexity
$\mathcal{AL}, \mathcal{ALN}$	PTIME
$\mathcal{ALU}, \mathcal{ALUN}$	NP-complete
$\mathcal{ALE}$	coNP-complete
$\mathcal{ALC}, \mathcal{ALCN}, \mathcal{ALCI}, \mathcal{ALCQI}$	PSPACE-complete

It can be observed that there are two sources of complexity:

- The union ($\mathcal{U}$) is of type NP.
- The existential quantification ($\mathcal{E}$) is of type coNP.

When they are combined, the complexity jumps to PSPACE.

Note that number restrictions ($\mathcal{N}$) do not add complexity.

9.4.3 Ontology Reasoning

The problem of **ontology satisfiability** consists of verifying whether an ontology is satisfiable or not, i.e., whether the ontology $\mathcal{O}$ admits at least one model.

The problem of **concept instance checking** consists of verifying whether an individual c is an instance of a concept C in $\mathcal{O}$, i.e., whether $\mathcal{O} \models C\,(c)$ or not.

The problem of **role instance checking** consists of verifying whther a pair (c_1, c_2) of individuals is an instance of a role R in $\mathcal{O}$, i.e., whether $\mathcal{O} \models R\,(c_1, c_2)$ or not.

The problem of **query answering** consists of finding the certain answers:

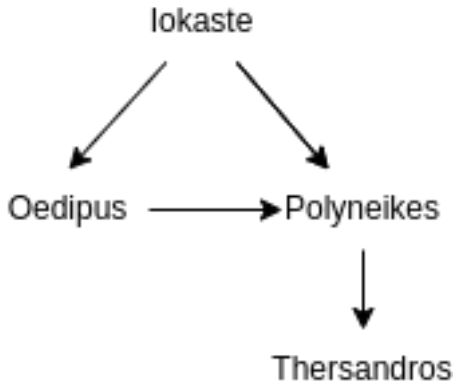
Definition 9.4. The **certain answers** to query $q\,(x)$ over $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, denoted $cert\,(q, \mathcal{O})$ are the tuples c of constants of $\mathcal{A}$ such that $c \in q^{\mathcal{I}}$, for every model $\mathcal{I}$ of $\mathcal{O}$.

We need to bear in mind the **open-world assumption**: something evaluates to false only if it contradicts other information in the ontology.

Example 9.7. Open-world assumption illustrative example.
Consider the following ABOX:

$$\begin{aligned} hasSon\,(Iokaste, Oedipus)\\ hasSon\,(Iokaste, Polyneikes)\\ hasSon\,(Oedipus, Polyneikes)\\ hasSon\,(Polyneikes, Thersandros)\\ patricide\,(Oedipus)\\ \neg patricide\,(Thersandros) \end{aligned}$$

This is visually shown as:



Consider the query:

$$Query \equiv \exists hasSon.\,(patricide \sqcap \exists hasSon.\neg patricide)$$

Does

$$ABOX \models Query\,(Iokaste)?$$

Yes!
Since there is no information of whether $patricide\,(Polyneikes)$ is true or not, we need to evaluate all possibilities.
If $patricide\,(Polyneikes)$ is true, then $hasSon\,(Iokaste, Polyneikes) \wedge patricide\,(Polyneikes) \wedge hasSon\,(Polyneikes, Thers$
$\neg patricide\,(Thersandros)$.
If $patricide\,(Polyneikes)$ is false, then $hasSon\,(Iokaste, Oedipus) \wedge patricide\,(Oedipus) \wedge hasSon\,(Oedipus, Polyneikes) \wedge$
$\neg patricide\,(Thersandros)$.

9.4.4 Modeling with Description Logics

It is hard to build good ontologies using description logics, because the names of the classes are irrelevant, classes are overlapping by default and the domain and range definitions are axioms, not constraints. In addition, we need to cope with the open world assumption and the non-unique name assumption, i.e., the same concept might be instantiated with two different names or IDs in the knowledge base[5].

10 Ontology Web Language (OWL)

OWL is a W3C recommendation, based on OIL and DAML, and using RDF and XML as the underlying representation. There were three languages in OWL 1.0: Lite, DL and Full; later, OWL 2.0 eliminated OWL Lite and added three profiles, RL, QL and EL.

[5]Some families, such as DL-Lite family, assume the unique name assumption.

10.1 OWL Axioms

OWL axiom	DL syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	Human⊑Animal⊓Biped
equivalentClass	$C_1 \equiv C_2$	Man≡Human⊓Male
disjointWith	$C_1 \sqsubseteq \neg C_2$	Man⊑ ¬Female
sameIndividualAs	$\{a_1\} \equiv \{a_2\}$	$\{presBush\} \equiv \{G.W.Bush\}$
differentFrom	$\{a_1\} \sqsubseteq \neg \{a_2\}$	$\{john\} \sqsubseteq \neg \{peter\}$
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter⊑hasChild
equivalentProperty	$P_1 \equiv P_2$	hasCost≡hasPrice
inverseOf	$P_1 \equiv P_2^-$	hasChild≡hasParent$^-$
transitiveProperty	$P^+ \sqsubseteq P$	ancestor$^+$ ⊑ancestor
functionalProperty	$T \sqsubseteq (\leq 1P)$	$Person \sqsubseteq (\leq 1\ hasFather)$
inverseFunctionalProperty	$T \sqsubseteq (\leq 1P^-)$	$Citizen \sqsubseteq (\leq 1\ hasSSN^-)$

10.2 OWL Constructs

OWL construct	DL syntax	Example
intersectionOf	$C_1 \sqcap ... \sqcap C_n$	Human⊓Male
unionOf	$C_1 \sqcup ... \sqcup C_n$	Doctor⊔Lawyer
complementOf	$\neg C$	¬Male
oneOf	$\{a_1\} \sqcup ... \sqcup \{a_n\}$	$\{john\} \sqcup \{mary\}$
allValuesFrom	$\forall P.C$	$\forall hasChild.Doctor$
someValuesFrom	$\exists P.C$	$\exists hasChild.Lawyer$
maxCardinality	$(\leq nP)$	$(\leq 1\ hasChild)$
minCardinality	$(\geq nP)$	$(\geq 2\ hasChild)$

Constructs such as owl:someValuesFrom, owl:allValuesFrom, owl:minCardinality, owl:maxCardinality are expressed using blank nodes, together with owl:Restriction by means of reification.

Example 10.1. Defining :Department $\sqsubseteq \forall$:Leads.:Professor

```
_:a rdfs:subClassOf owl:Restriction
_:a owl:onProperty :Leads
_:a owl:allValuesFrom :Professor
:Department rdfs:subClassOf _:a
```

Here:

- rdfs:subClassOf owl:Restriction indicate that _:a is a complex constraint
- owl:onProperty indicates the constrained property (:Leads in this case)
- owl:allValuesFrom denotes the qualified constraint or cardinality

Example 10.2. Defining :student $\sqsubseteq \geq$ 3:RegisteredTo and :student $\sqsubseteq \leq$ 6:RegisteredTo

```
_:a rdfs:subClassOf owl:Restriction
_:a owl:onProperty :RegisteredTo
_:a owl:minCardinality 3
_:b rdfs:subClassOf owl:Restriction
_:b owl:onProperty :RegisteredTo
_:b owl:maxCardinality 6
:Student rdfs:subClassOf _:a
:Student rdfs:subClassOf _:b
```

Example 10.3. Defining $C_1 \sqsubseteq \exists P.C$

```
_:a rdfs:subClassOf owl:Restriction
_:a owl:onProperty :P
_:a owl:someValuesFrom C
C1 rdfs:subClassOf _:a
```

Example 10.4. Defining $\exists : TeachesTo. : Undergrad \sqsubseteq : Professor \sqcup : Lecturer$

```
_:a rdfs:subClassOf owl:Restriction
_:a owl:onProperty :TeachesTo
_:a owl:someValuesFrom :Undergrad
_:b owl:unionOf (:Professor,:Lecturer)
_:a rdfs:subClassOf _:b
```

10.3 OWL Implementation

OWL uses RDF syntax, i.e., URIs and literals that conform valid triplets. It reuses some URIs from RDFS, but the whole RDFS is not embedded in OWL. In addition, it adds new properties and classes based on description logics and defined at the OWL namespace.

10.4 OWL 2.0 Profiles

- OWL 2 EL: based in EL++, suitable for knowledge bases with large numebr of properties and classes. The reasoning is polynomial with respect to the size of the TBOX.
- OWL 2 QL: based on DL-Lite, captures ER and UML expressive power. The reasoning is reducible to LOGSPACE.
- OWL 2 RL: based on description logic programs, provides scalable reasoning without sacrificing much expressivity. The reasoning is polynomial with respect to the size of the ontology.

11 Graph-Based Virtual Data Integration

Data Integration is an area of study within data management aiming at facilitating transparent acces to a variety of heterogeneous data sources.

There are two main ways to perform data integration:

- Physical data integration: in physical data integration, the data is integrated by explicit and real modification and movement of the data, storing the transformed data in an integrated database.
- Virtual data integration: in this case, the approach is to encapsulate the physical data with views, with these being those that are integrated and queried. Therefore, in virtual data integration, data remains at the origin, and it is provided to the system transparently by means of views.

Virtual Data Integration is suitable when data sources are not under our control, and the owners require a federation, when we do not want to move the data from where it resides, when data freshness is crucial (ETLs run from time to time and the period between updates is called the **update window**), and when we want to systematically trace all the data available within a company or organization (**data provenance**).

There is consensus that graph-based solutions are the way to go for data integration and governance, because graph-based data models are richer than any other data model and can express any construct, with knowledge graphs preferred over property graphs because they facilitate linking TBOX/ABOX with little effort. In addition, they allow to represent all data integration constructs with the same formalism: target schema + source schema + mappings.

There are two main approaches to achieve data integration:

- Ontology-based data access: this is a monolithic approach, in which the TBOX is directly related to the sources via mappings.
- Ontology-mediated queries: this approach relies on the concept of **wrapper**, which is a view presenting the source data. Thus, it allows to select a subset of the data source to be exposed to the whole integration system, allowing for security and modularity, as well as pay-as-you-go data integration, i.e., building the integrated schema incrementally as new data sources arrive.

11.1 Local-As-View (LAV) Integration - Ontology-Mediated Queries (OMQ)

OMQ us a family of systems performing graph-based data integration with LAV (and conceptually GAV is also possible). It is based on the well-known **wrapper-mediator architecture.**

To make the querying rewriting feasible, they adopt several measures:

- Exact mappings (closed-world assumption).
- Very basic reasoning capabilities: only taxonomies and domain/range inference.

11.1.1 Big Data Integration Ontology

The BDIO revisits the Data Integration framework and construcs an ontology as follows:

- It defines the **global level**, G, which is the integrated view of the data.
- It defines the **source levels**, S, which are views (wrappers) on the data sources.
- And it defines the **mappings**, M, which are the LAV mappings between G and S.

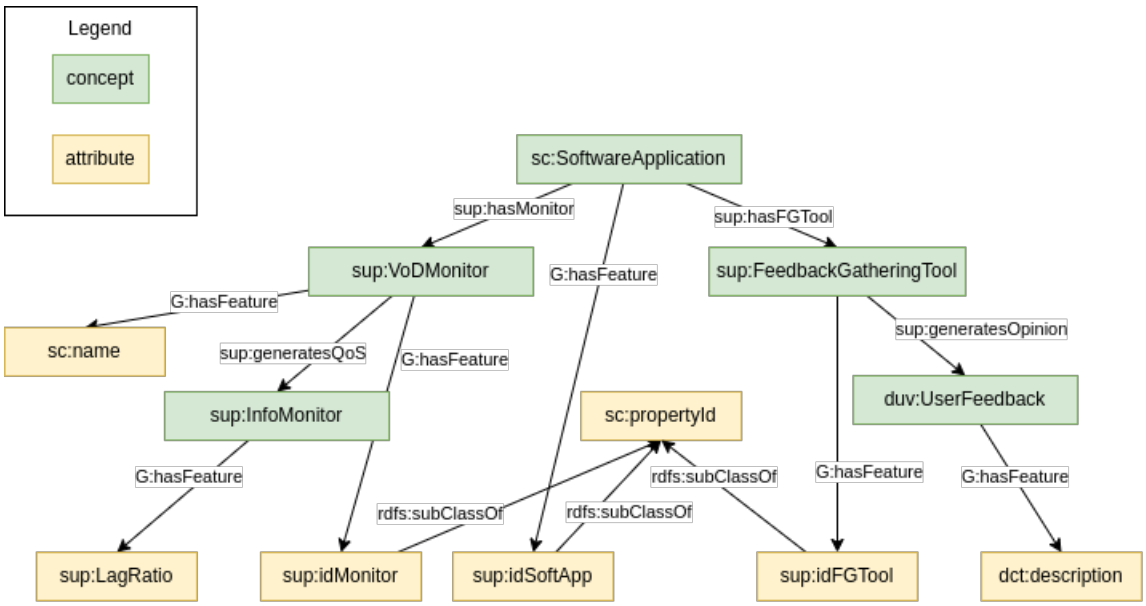
A **wrapper** represents a view on the source, and we can think of it as a named query over the source (a view).

Some typical assumptions made by wrappers are:

- They expose the source data in tabular format (1NF).
- A data source may generate several wrappers.
- Typically, new versions of data are considered new wrappers.

Example 11.1. A LAV Integration

First, we define the global level, G:



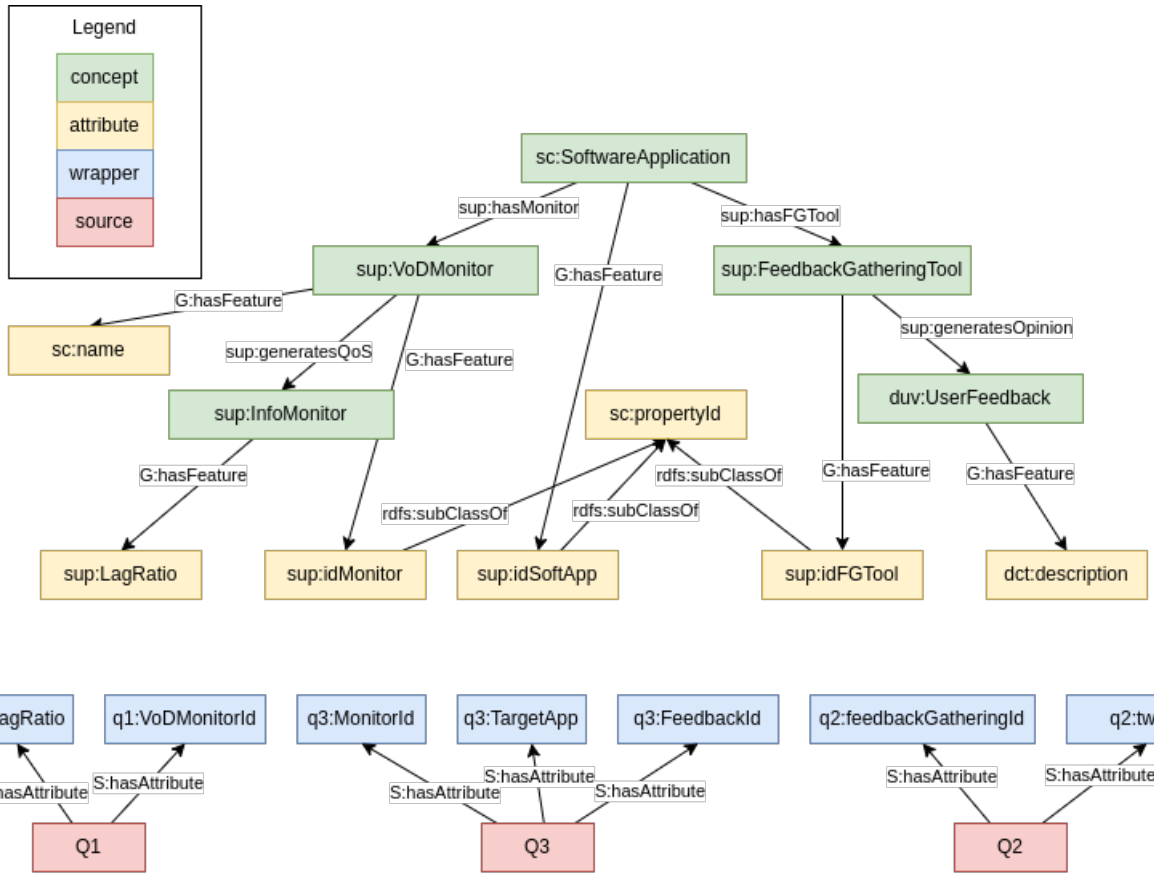
Now, we need to expose the sources by means of wrappers. For this, we automatically bootstrap the attributes projected by the wrappers.

For instance, let's say our data comes from three sources, Q1, Q2 and Q3:

```
Q1: ID and compute the lag ratio
  db.getCollection('vod').aggregate([
    {$project: {"VoDMonitorId": true,
          "lagRatio": {$divide: ["$waitTime", "$watchTime"]}
          }
    }
  )]
```

```
Q2: All attributes for tweets in English.

Q3: association target app -> monitor, feedback gathering tool
```

We end up with the following:



Finally, we need to define the LAV mappings for the wrappers. A LAV mapping for a wrapper Q is defined as $M = \langle G, S \rangle$ where G is a named graph and S is a set of triples of the form:

<x, owl:sameAs, y>

where <x, rdf:type, S:Attribute> and <y, rdf:type, G:Feature>.

In our example, we can define the LAV mapping for the wrappers for source Q1:

```
-- First we define G, the named graph
Q1 S:provides {
        sup:InfoMonitor G:hasFeature sup:lagRatio .
        sup:VoDMonitor sup:generatesQoS sup:InfoMonitor .
        sup:VoDMonitor G:hasFeature sup:idMonitor
      }

-- Now we define S, the "same as" triples
q1:lagRatio owl:sameAs sup:lagRatio
q1:VoDMonitorId owl:sameAs sup:idMonitor
```

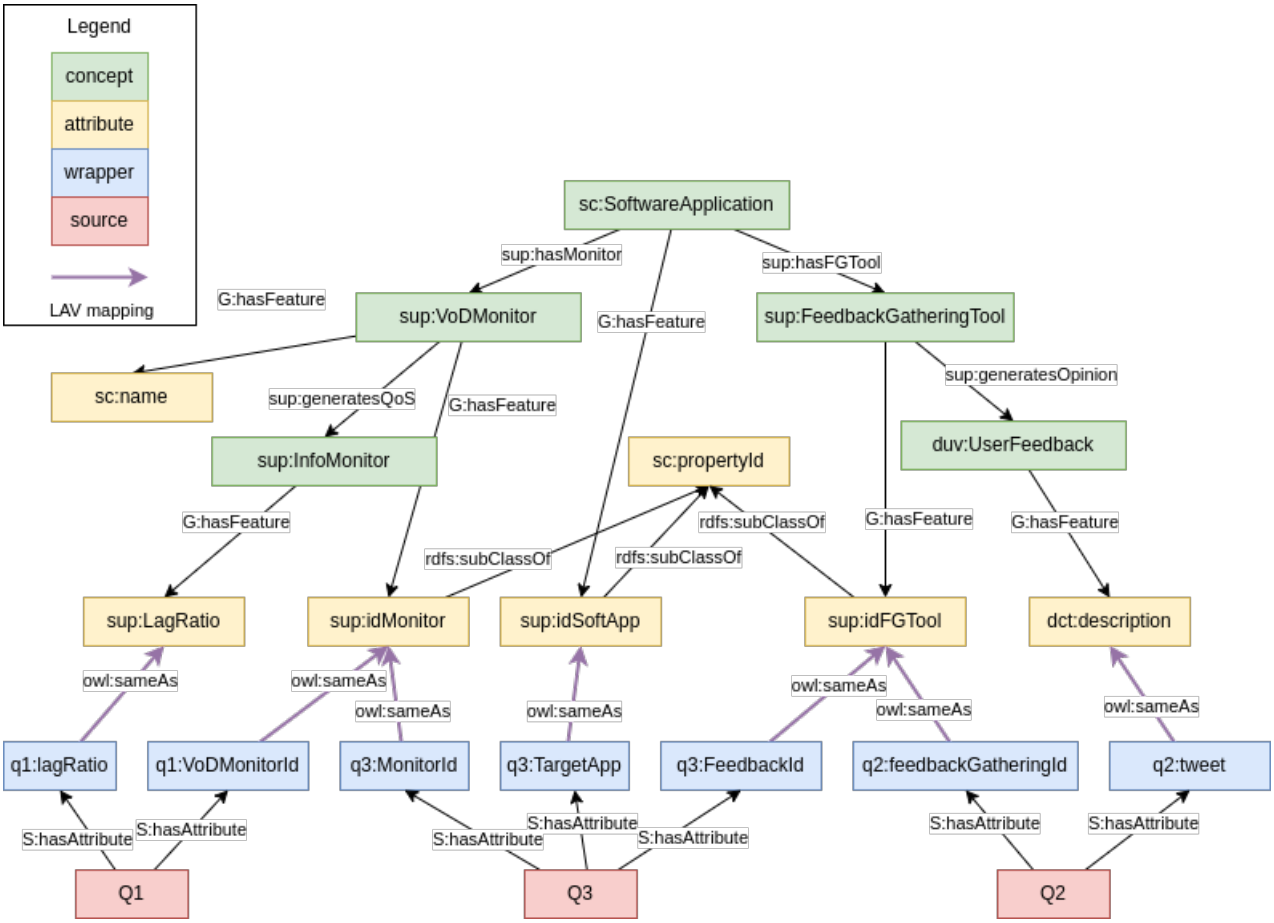
For source Q2:

```
Q2 S:provides {
        sup:FeedbackGatheringTool sup:generatesOpinion duv:UserFeedback .
        sup:FeedbackGatheringTool G:hasFeature sup:idFGTool .
        duv:UserFeedback G:hasFeature dct:description
      }

q2:feedbackGatheringId owl:sameAs sup:idFGTool
q2:tweet owl:sameAs dct:description
```

And for source Q3:

```
Q3 S:provides {
        sc:SoftwareApplication sup:hasMonitor sup:VoDMonitor .
        sc:SoftwareApplication sup:hasFGTool sup:FeedbackGatheringTool .
        sc:SoftwareApplication G:hasFeature sup:idSoftApp .
        sup:VoDMonitor G:hasFeature sup:idMonitor .
        sup:FeedbackGatheringTool G:hasFeature sup:idFGTool
      }

q3:MonitorId owl:sameAs sup:idMonitor
q3:TargetApp owl:sameAs sup:idSoftApp
q3:FeedbackId owl:sameAs sup:idFGTool
```

And we end up with the following integrated view of the data:



11.2 Query Answering - Rewriting Algorithm

Any SPARQL query on the global graph must be rewritten as a query in terms of the wrappers, which provide the way to access the actual data.

For example, assume the following SPARQL query:

```
SELECT ?w, ?t WHERE
  ?t rdf:type sup:lagRatio .
  ?x G:hasFeature ?t .
  ?x rdf:type sup:InfoMonitor .
  ?y sup:generatedQoS ?x .
  ?y rdf:type sup:VoDMonitor .
  ?z sup:hasMonitor ?y .
  ?z rdf:type sc:SoftwareApp .
  ?z G:hasFeature ?w .
  ?w rdf:type sup:idSoftwareApp .
```

```
11 FILTER ?w = "SUPERSEDE"
```

We start from terminal features. In this case, sup:lagRatio which maps to q1:lagRatio:

$$\Pi_t \left(\rho_{q1:lagRatio \to t} \left(Q1 \right) \right),$$

then, we go up the ontology, reaching sup:InfoMonitor. The query does not change at this point because this is schema information covered by Q1. We continue going up, until sup:VoDMonitor, and the query still remains the same. Now, we reach sc:SoftwareApplication, which is not covered by Q1, but by Q3. Therefore, we need to join them somehow. Q1 and Q3 share sup:idMonitor, so we join them using this attribute:

$$\Pi_t \left(\rho_{q1:lagRatio \to t} \left(\sigma_{q1:VoDMonitorId = q3:MonitorId} \left(Q1 \times Q3 \right) \right) \right),$$

we continue reading the query, reaching sup:idSoftwareApp, which is another feature that needs to be added to the query:

$$\Pi_{w,t} \left(\rho_{q1:lagRatio \to t} \rho_{q3:targetApp \to w} \left(\sigma_{q1:VoDMonitorId = q3:MonitorId} \left(Q1 \times Q3 \right) \right) \right),$$

finally, we have to apply the filter

$$\sigma_{w = "SUPERSEDE"} \left(\Pi_{w,t} \left(\rho_{q1:lagRatio \to t} \rho_{q3:targetApp \to w} \left(\sigma_{q1:VoDMonitorId = q3:MonitorId} \left(Q1 \times Q3 \right) \right) \right) \right).$$

11.2.1 Computational Complexity

This query rewriting algorithm is linear in the size of the subgraph of G to navigate, linear in the size of the wrappers mappings and exponential in the number of wrappers that may join. The good thing is that evidence shows that typically BigData sources have few join points and therefore the exponential complexity is affordable in real cases.

References

[1] Óscar Romero and Anna Queralt. Semantic data management. Lecture Notes.