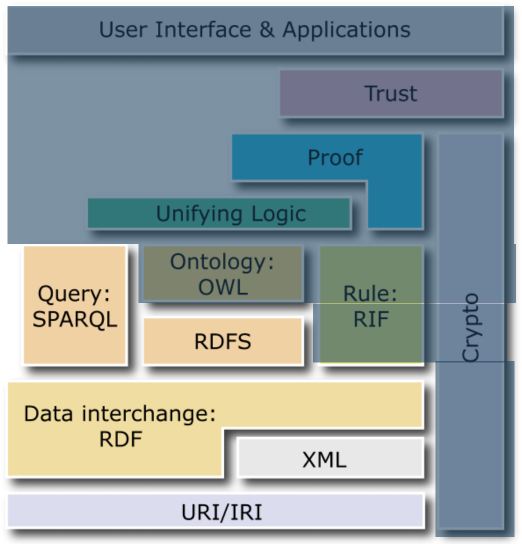
SPARQL

Querying the Semantic Web



FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

How to query RDF datasets?

- Now that we have data in RDF(S), we should be able to query and update it
- Using RDF(S) semantics could be an alternative
  - If we wanted to know whether `ex:jja` is a professor, we could check if the following triple is a consequence of the graph

    ```
    ex:jja rdf:type uni:Professor
    ```

  - Or, if we want to know who are the professors, we could check which triples of the form `X rdf:type uni:Professor` belong to the closure of the RDF graph

Querying RDF with RDFS

- How far could we take the approach?
  - Ask whether a given triple is a consequence
  - Ask for the resources that are instances of a class
  - Ask whether a class subsumes another class

- For example:
  - Who are the associate or full professors?
    - Define a new class `uni:ProfOrAssoc`, as a superclass of `uni:Professor` and of `uni:Associate`, and then ask for the instances of that new class
  - Who are the ones who teach something?
    - Define a new class `uni:teacher`, and add that the `rdfs:domain` of `ex:teaches` is (also) `uni:teacher`. Then ask for the instances of this new class

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Querying RDF with RDFS

- What about:
  - Who are the professors who teach something?
  - Who are the professors older than 50 years?
  - Who are the students involved in courses taught by ex:jja?
  - Who are the pairs of students who share a common professor at any of the courses they are involved in?
- RDFS is not expressive enough!
  - It is not possible to define classes who gather all the individuals that satisfy the conditions in the above queries
- A more expressive query language is needed.

Querying RDF/XML

- What if we use XML query languages, and directly query the RDF/XML encoding?
  - One can just view the RDF as an XML file, and use XPath, or XQuery
- Who are the Professors who teach something?

```
<rdf:Description rdf:id="sw">
  <ex:courseName> Semantic Web </ex:courseName>
</rdf:Description>
<rdf:Description rdf:id="jja">
  <ex:name> José Alferes </ex:name>
  <rdf:type rdf:resource="uni:Professor"/>
</rdf:Description>
<rdf:Description rdf:about="#jja">
  <ex:teaches rdf:resource="#sw"/>
</rdf:Description>
```

Querying RDF/XML

- What if we use XML query languages, and directly query the RDF/XML encoding?
  - One can just view the RDF as an XML file, and use XPath, or XQuery
- Who are the Professors who teach something?

```
<rdf:Description rdf:id="sw" ex:courseName="Semantic Web"/>
<uni:Professor rdf:id="jja" ex:name = "José Alferes">
  <ex:teaches rdf:resource="#sw"/>
</uni:Professor>
```

Querying RDF/XML

- What if we use XML query languages, and directly query the RDF/XML encoding?
  - One can just view the RDF as an XML file, and use XPath, or XQuery

- This would be totally dependent on the syntactic representation
  - (which, even worse, is not unique)
- It would defeat the whole purpose of RDF
  - It would be totally useless!

SPARQL Query Language

- SPARQL - Sparql Protocol And Rdf Query Language (read as *sparkle)* provides facilities to:
  - extract information in the form of URIs, blank nodes, plain and typed literals.
  - extract RDF sub-graphs.
  - construct new RDF graphs based on information in the queried graphs.
- RDF graphs can be obtained from several sources, including middleware capable of generating RDF data.
- SPARQL is based on matching of graph patterns, using variables in the patterns
  - Results can come as binding of variables to RDF terms (URIs, literals or blank nodes) in a tabular form

Status of development

- First W3C recommendation in February 2008
- Extension to SPARQL 1.1 as of March 2013
  - Query language
  - Update language (insertion and deletion)
  - HTTP operations to manage collections of graphs
  - Entailment regimes
    - simple entailment is the default, but RDF entailment, RDFS entailment, and others to be seen, are possible
  - Federation extensions (distributed queries)
  - SPARQL Service description
  - Query results in several formats: XML, JSON, CSV, …

Query forms

- SPARQL has four query forms:
  - **`SELECT`**, returns variable bindings
  - **`ASK`**, returns a boolean indicating whether a query pattern matches or not
  - **`DESCRIBE`**, returns a RDF graph that describes the resources found
  - **`CONSTRUCT`**, returns RDF graphs by substituting variables in a set of triple templates

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Simple SELECT query

- The query consists of two parts, the `SELECT` clause and the `WHERE` clause.
  - The `SELECT` clause identifies the variables to appear in the query results.
  - The `WHERE` specifies the graph pattern to match against the RDF data graph.
- The simplest form of graph patterns are triple patterns
  - a triple pattern is an RDF triple with optional query variables in any place of the triple.
- Basic Graph Patterns (BGPs) are sets of triple patterns, in Turtle syntax

Our first SPARQL query

```
PREFIX uni: <http://fct.unl.pt/concepts/>
PREFIX  ex: <http://fct.unl.pt/example/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x ?name
WHERE {?x ex:teaches ex:sw .
       ?x rdf:type uni:Professor .
       ?x ex:name ?name }
```

- Professor who teach `ex:sw`, and their names
- Syntax:
  - Abbreviated URIs with PREFIX
  - Variable names signalled by ? (or by $)
  - Literals are as in turtle
  - ; and , can be used as in Turtle

Our first SPARQL query

```
…
SELECT ?x ?name
WHERE {?x ex:teaches ex:sw ;
          rdf:type uni:Professor ;
          ex:name ?name }
```

- Professors who teach ex:sw, and their names

- Determines the parts of the graph that match the BGP in the `WHERE` clause
- Returns the bindings of variables in the `SELECT` clause

?x	?name
`<http://fct.unl.pt/example/jja>`	`"Jose Alferes"`

Variables everywhere!

- Variables are allowed in subject, object and also in predicate positions of triples
  - E.g. what are the relations between a guy called José Alferes and the object ex:sw?

```
…
SELECT ?relation
WHERE {?x ex:name   "Jose Alferes" .
       ?x ?relation ex:sw }
```

?relation
<http://fct.unl.pt/example/teaches>

Blank nodes

- Can be used in query patterns
  - As in RDF, cannot be used in predicates
  - Act like variables, but cannot be selected
    - Remember the semantics of bnodes in RDF
  - Have arbitrary ID, and cannot be reused in different BGP
- Can be returned in results (if they are bnodes in the original graph)
  - Placeholders for unknown elements
  - With arbitrary IDs, with the scope limited to the result, that can be different from those in the input RDF
    - But repeated occurrences in the result denote the same element

```
SELECT ?name
WHERE {_:a ex:teaches ex:sw ;
           rdf:type uni:Professor ;
           ex:name ?name }
```

Datatypes

- Exact match for the datatypes is required
  - E.g. `{?x ex:p "test" .}` does not match with `ex:ex1 ex:p "test"^^xsd:string .`
- But abbreviation for numbers are allowed
  - E.g. `{ ?x ex:p 123 .}` matches with `ex:ex1 ex:p "123"^^xsd:integer .`
  - The datatype is determined from the syntax
    - `xsd:integer` (e.g. 123)
    - `xsd:decimal` (e.g. 123.45)
    - `xsd:double` (e.g. 1.23e2)

Some syntactic simplifications

- Collections
  - RDF collections can be written using the syntax `( element1 element2 … )`
  - `rdf:nil` can be replaced by `()`
  - E.g. `{(1 ?x) ex:p "w"}` is the same as
    ```
    {_:a rdf:first 1 ; rdf:rest _:b .
    _:b rdf:fist ?x; rdf:rest rdf:nil .
    _:a ex:p "w"}
    ```
- `rdf:type`
  - Can be replaced by a. E.g. `{?x a uni:Professor}`
  - Reads better…

RDF Datasets

- When compared with SQL, what is missing?
  - The FROM clause
- No FROM clause is required in SPARQL
  - A SPARQL query is executed against an (implicit) RDF Dataset which represents a collection of graphs
  - RDF data stores hold multiple RDF graphs, and record information about each graph, allowing an application to make queries that involve information from more than one graph (a collection of graphs)
- But more (named) graphs can be specified

Graph provenance

- SPARQL can use the keywords `FROM` and `FROM NAMED`, to specify the default graph and named graphs, respectively.
- The `FROM` and `FROM NAMED` are followed by an URI.
- The graphs retrieved by `FROM` clauses are merged for constructing the default graph
- The `FROM NAMED` construct allows to include in the RDF dataset a named graph.
- The `GRAPH` construct can be used to obtain the provenance of results, or to query a particular named graph. Outside the `GRAPH` construct, the queries are always relative to the default graph.

Provenance example

```
SELECT ?graph ?name ?mbox
FROM NAMED <http://ex.org/a>
FROM NAMED <http://ex.org/b>
WHERE {
   GRAPH ?graph
     { [ foaf:name ?name; foaf:mbox ?mbox ] } }
```

?graph	?name	?mbox
<http://ex.org/a>	Jose	<mailto:j@example.pt>
<http://ex.org/a>	Maria	<mailto:m@example.pt>
…	…	…
<http://ex.org/b>	Ana	<mailto:a@example.pt>
<http://ex.org/b>	Jose	<mailto:j@example.pt>
…	…	…

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

Provenance example

```
SELECT ?name ?mbox
FROM NAMED <http://ex.org/a>
FROM NAMED <http://ex.org/b>
WHERE {
   GRAPH <http://ex.org/a>
     { [ foaf:name ?name; foaf:mbox ?mbox ] }
   GRAPH <http://ex.org/b>
     { [ foaf:name ?name; foaf:mbox ?mbox ] }
 }
```

?name	?mbox
Jose	<mailto:j@example.pt>

Multiple BGP and UNION

- BGP can be grouped, and several can be used.
- `{}` stand for the empty BGP, and matches any graph
- `UNION` between BGPs stand for the union (disjunction) of the BGPs.
  - E.g. Who are the associate or full professors

```
SELECT ?x
WHERE {
    { ?x a uni:Professor }
    UNION
    { ?x a uni:Associate }
  }
```

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

Unbound results

- `UNION` may give rise to unbound results.
  - E.g. What are the names or nomes of courses?

```
SELECT ?x ?name ?nome
WHERE {
 ?x a uni:Course .
 { ?x ex:name ?name } UNION { ?x ex:nome ?nome}
}
```

?x	?name	?nome
ex:sw	"Semantic Web"	
ex:sw		"Web Semântica"@pt
ex:sbd		"Sistemas de BD"@pt

OPTIONAL patterns

- Another keyword that may return unbound results is `OPTIONAL`
- It allows for the specification of optional parts of a graph
  - E.g. What are the nomes of courses, and names if they exist?

```
SELECT ?x ?name ?nome
WHERE {
 ?x a uni:Course . ?x ex:nome ?nome
 OPTIONAL {?x ex:name ?name} }
```

?x	?name	?nome
ex:sw	"Semantic Web"	"Web Semântica"@pt
ex:sbd		"Sistemas de BD"@pt

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

Combination of UNION and OPTIONAL

- When used in combination UNION has higher precedence
  - I.e. X `UNION` Y `OPTIONAL` Z is to be interpreted as `{`X `UNION` Y`}` `OPTIONAL` Z
  - E.g. The names of either authors or editors of `ex:swBook`, and also the email when available?

```
SELECT ?name ?email
WHERE {
    ?x ex:name ?name .
  { {ex:swBook ex:author ?x }
    UNION
    {ex:swBook ex:edit ?x } }
    OPTIONAL { ?x ex:email ?email} }
```

Filters

- Even with complex graph patterns, some quite reasonable queries are not expressible:
  - Who are the professors aged more than 50?
  - Professors with a name started by "A"?
- `FILTER` allows us to specify conditions over variables, restricting the solutions to those that satisfy the condition

```
SELECT ?prof
WHERE {
   ?prof a uni:Professor; ex:age ?age .
   FILTER ( ?age > 50 )
 }
```

Filter conditions

- Conditions evaluate to truth values (booleans)
- Usual comparison operator are allowed
  - = and != for all RDF types
  - >, <, <= and >= for numerical datatypes, dates, string and booleans
- Function can be used
  - Arithmetic function (+, -, *, /) for numerical datatypes
    - e.g. `FILTER { ?x / ?y > ?z }`
  - XQuery and XPath functions
  - Specific SPARQL functions

Specific functions

`BOUND(A)`	returns true if A is a bound variable
`ISURI(A)`	returns true if A is (bound to) a URI
`ISBLANK(A)`	returns true if A a bnode
`ISLITERAL(A)`	returns true if A a literal
`SAMETERM(A,B)`	returns true if A and B are the same RDF term
`REGEX(A,B)`	returns true if the string A matches the regular expression B
`DATATYPE(A)`	returns the datatype of A
`LANG(A)`	returns the language tag of A (or "" if none exists)
...	...

- Several more functions, especially in SPARQL 1.1
  - Check the manual/specification

Combining conditions

- Conditions can be combined
  - Conjunctively, by adding multiple filters
  - Disjunctively, with `UNION`
- But boolean operators `&&`, `||` and `!` are also available
  - E.g. Professors between 20 and 30, or not younger than 50?

```
SELECT ?prof
WHERE {
   ?prof a uni:Professor; ex:age ?age .
   FILTER( (?age > 20 && ?age < 30 ) ||
            !(?age < 50) )
 }
```

Solution modifiers

- The result of a `SELECT` query is a bag of answers (a table)
  - But one can view it as a sequence of tuples, and consider their order
  - Or as a set, without repeated elements
- Similarly to SQL, there can be an `ORDER BY` clause, and select only `DISTINCT` values
  - By default the order is ascending
  - For descending, use `DESC`
  - String and numbers are sorted as usual
  - URIs are sorted as strings
  - unbound variables < bnodes < URIs < Literals
- One can also `LIMIT` the number of results, and define an `OFFSET` for the first result shown

Modifiers examples

- 10 oldest professors sorted by age, from oldest to younger:

```
SELECT ?prof ?age
WHERE { ?prof a uni:Professor; ex:age ?age}
ORDER DESC(?age)
LIMIT 10
```

- Second oldest professor:

```
SELECT ?prof ?age
WHERE { ?prof a uni:Professor; ex:age ?age}
ORDER DESC(?age)
LIMIT 1 OFFSET 2
```

- Ages of professors

```
SELECT DISTINCT ?age
WHERE { [] a uni:Professor; ex:age ?age}
```

Aggregates

- SPARQL 1.1 includes aggregates and aggregate functions
  - As in SQL, aggregates group solutions and compute values over the groups
  - Groups are defined by a `GROUP BY` clause
  - There can be filters over groups, with `HAVING`
- Available aggregate function:
  - `COUNT`, `MIN`, `MAX`, `SUM`, `AVG` are as usual in SQL
  - `SAMPLE`, picks one random value from the group
  - `GROUP_CONCAT`, concatenate strings in the group

Aggregates examples

- How many courses does each lecturer who teach more than 2, teach

```
SELECT ?prof COUNT(?course) AS ?ncourses
WHERE { ?prof ex:teaches ?course }
GROUP BY ?prof
HAVING ?ncourses > 2
```

  - Note the renaming of results with `AS`

- String with list of professors' names by age

```
SELECT ?age GROUP_CONCAT(?name, separator=", ")
WHERE { _:x a        uni:Professor;
            ex:age   ?age;
            ex:name ?name
      }
GROUP BY ?age
```

Subqueries

- In SPARQL 1.1 one can have subqueries, whose results are conjunctively joined with the remainder BGP
  - E.g. name of courses that are taught by exactly two lecturers

```
SELECT ?cname
WHERE {
        ?course ex:name ?cname .
        { SELECT ?course COUNT(?lect) AS ?nl
          WHERE { ?lect ex:teaches ?course }
          GROUP BY ?course
          HAVING ?nl = 2
        }
     }
```

Variable BINDing

- In SPARQL 1.1 variables can be bound to expressions with `BIND` and `AS`

```
SELECT ?name ?yearBirth
WHERE {
        [] a uni:Professor; ex:age ?age
        BIND ((2019 - ?age) AS ?yearBirth)
     }
```

- Or

```
SELECT ?name (2019 - ?age) AS ?yearBirth
WHERE { [] a uni:Professor; ex:age ?age }
```

Property paths

- Another feature of SPARQL 1.1 is property paths
  - Extend patterns from simple triples to paths, eventually with arbitrary length
- Property paths are built with regular expressions over predicates
  - Alternative paths: `?s (exp1 | … | expn) ?o`
  - Inverse paths: `?s ^exp ?o` (the same as `?o exp ?s`)
  - Negation of paths: `?s !exp ?o`
  - Sequence of paths: `?s exp1/…/expn ?o`
  - Paths with arbitrary length: `?s exp+ ?o`, `?s exp* ?o`, and optional presence `?s exp? ?o`
  - Paths with given length limits: `?s exp{n} ?o`, `?s exp{n,} ?o`, and `?s exp{,n} ?o`, `?s exp{n,m} ?o`

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

Property paths example

- The ancestors of `ex:jja`:

```
SELECT ?ans
WHERE { ex:jja (ex:mother|ex:father)+ ?ans }
```

- The descendants of `ex:jja`:

```
SELECT ?desc
WHERE { ex:jja (^ex:mother|^ex:father)+ ?desc }
```

- The classes to which `ex:jja` belongs:

```
SELECT ?class
WHERE { ex:jja rdf:type/rdfs:subClassOf* ?class}
```

- The object to which `ex:jja` is related to, except for its type:

```
SELECT ?o
WHERE { ex:jja !(rdf:type) ?o }
```

Negation

- SPARQL 1.1 includes two forms of negation, besides the negation in filter conditions
  - Test for patterns that do **not** exist
  - Set difference between patterns
- The first with `FILTER NOT EXISTS` {BGP}
- The second with `MINUS` between BGPs

Negation examples

- Professors for which there is no age

```
SELECT ?prof
WHERE {
   ?prof a uni:Professor .
   FILTER NOT EXISTS {?prof ex:age _:a}
 }
```

- Can also be done with MINUS

```
SELECT ?prof
WHERE {
   ?prof a uni:Professor .
   MINUS {?prof ex:age ?age}
 }
```

Testing for existence

- If instead of wanting to know the results (variable bindings) one is only interested in knowing whether there are results, we can simply use `ASK`
- Results of `ASK` are yes or no
- E.g. Is any lecturer of `ex:sw` called Jose Alferes?

```
ASK
WHERE {
   ?prof ex:name "Jose Alferes";
         ex:teaches ex:sw
 }
```

Returning graphs

- `SELECT` takes a (set of) RDF graph(s), and returns a table
  - In this sense, it is not algebraic
- SPARQL includes two commands that are algebraic
  - `DESCRIBE`: given a set of URIs returns the graph with all edges departing from that URI
  - `CONSTRUCT`: Similar to `SELECT`, but returning the result in the form of an RDF graph

Describe

- If given just a URI, returns all the direct data about the resource. E.g.
  - `DESCRIBE <http://dbpedia.org/resource/Portugal>`
- It can also be used to describe the result of a query that returns just one variable, which is always bound to URIs
  - E.g. The data about those that teach `ex:sw`

```
DESCRIBE ?prof
WHERE { ?prof ex:teaches ex:sw }
```

Construct

- `CONSTRUCT` constructs an RDF graph from a given graph template, using the variables of a query
- E.g. graph with name (as nome) and year of birth of lecturers of `ex:sw`

```
CONSTRUCT { ?prof my_ex:nome ?name;
                  my_ex:birth ?year }
WHERE { ?prof ex:teaches ex:sw;
              ex:age ?age; ex:name ?name
        BIND ((2019-?age) AS ?year)}
```

- E.g. all data from lecturers of `ex:sw`

```
CONSTRUCT { ?prof ?prop ?obj }
WHERE { ?prof ex:teaches ex:sw; ?prop ?obj }
```

SPARQL Updates

- SPARQL update make it possible to
  - insert and delete contents to graph stores
  - manage the lifecycle of graph stores
  - Somehow, similar to DML of SQL
- `INSERT` and `DELETE` insert or delete a given set of triple into a store
  - Like in `CONSTRUCT`
  - Delete cannot be applied to bnodes

```
INSERT { ?person ex:birth ?year }
DELETE { ?person ex:age ?age}
WHERE { ?person a ex:Person; ex:age ?age .
        BIND ((2019-?age) AS ?year)}
```

Managing graph stores

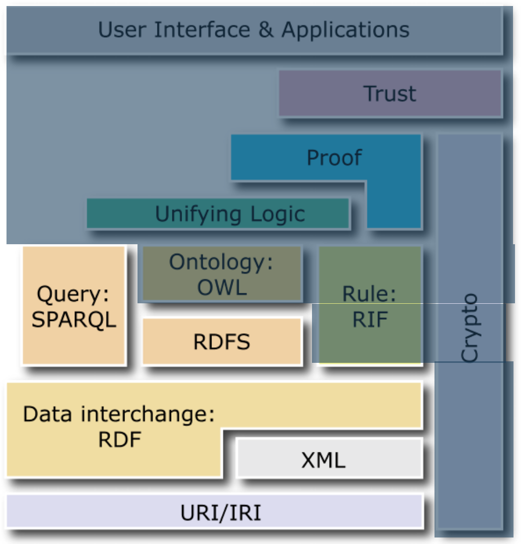
- There are extra constructs to manage RDF stores
  - `LOAD <URI1> [INTO GRAPH <URI2>]` loads all the triples in `URI1` to the store [in `URI2`]
  - `CLEAR [GRAPH <URI>]` deletes all the triples in the store [in `URI`]
  - `COPY GRAPH <URI1> TO <URI2>` copies and overwrites
  - `MOVE GRAPH <URI1> TO <URI2>` moves and overwrites
  - `CREATE GRAPH <URI>`
  - `DROP GRAPH <URI>`

Summary and follow-up

- SPARQL is a very powerful query language for RDF graphs
  - More expressive than RDFS
  - With some constructs that resemble SQL
  - It allows for effective querying of RDF data in the Semantic Web
- But what exactly is the meaning of a SPARQL query?
  - We've seen that informally, but we should look for the formal (exact) semantics of SPARQL - via SPARQL algebra
  - And how can one implement a SPARQL endpoint?
    - We must have a look at the SPARQL protocol

SPARQL Semantics

Meaning of Queries in the Semantic Web



Meaning of queries

- What is the exact meaning of a SPARQL query?
  - Up to now we've seen it informally
  - But to really use it, and to implement it, "informally" is not enough!
- How to define the meaning of a query language?
  - **Query Entailment** (as seen, e.g., in RDF(S))
    - Data described / translated into some formal logical language
    - Queries as possible consequences (of a given format)
    - Results as logical entailment
  - **Query Algebra** (as, e.g., in SQL)
    - Queries as algebraic expressions that compute the result, given the data as input.

SPARQL Algebra

- Unlike SQL, SPARQL is not directly an algebraic language
  - Each SQL query operates on relations, and returns a relation
  - SPARQL queries operate on RDF graphs and returns solutions
    - The CONSTRUCT doesn’t help here (in the middle we have relations)
- So, first we need to transform SPARQL queries into some algebra expression
  - The semantics is provided by evaluation of such transformed expressions

Transformation general idea

```
SELECT ?x ?name
WHERE {?x ex:teaches ex:sw ; ex:age ?age.
       ?x rdf:type uni:Professor .
       OPTIONAL {?x ex:name ?name}
       FILTER (?age < 50) }
```

- First take care of syntactic issues (and expand URIs)
- Take a Bgp(.) operator that given a pattern triple, returns a relation with assignment of variables
  - Apply it to each individual pattern triple in the query
- Have algebraic operator on relations for inner joins , outer joins, unions, filters, etc
  - translate the query by appropriate use of these operators

Transformation by example

- First take care of syntactic details (sorry but I won't expand rdf)

```
{?x <http://fct.unl.pt/example/teaches>
    <http://fct.unl.pt/example/sw> .
 ?x <http://fct.unl.pt/example/age> ?age.
 ?x rdf:type <http://fct.unl.pt/concepts/Professor> .
 OPTIONAL {?x <http://fct.unl.pt/example/name> ?name}
 FILTER (?age < 50) }
```

- Then apply Bgp(.) to triples (sorry but, for readability, I'll use prefixes anyways)

```
{Bgp(?x ex:teaches ex:sw)
 Bgp(?x ex:age ?age)
 Bgp(?x rdf:type uni:Professor)
 OPTIONAL {Bgp(?x ex:name ?name)}
 FILTER (?age < 50) }
```

Transformation by example

- Apply Join(.) for consecutive (groups of) Bgp

```
 {Join( Bgp(?x ex:teaches ex:sw),
         Join(Bgp(?x ex:age ?age),Bgp(?x rdf:type uni:Professor))
  OPTIONAL {Bgp(?x ex:name ?name)}
  FILTER (?age < 50) }
```

- Apply Leftjoin(.) for OPTIONALs

```
{LeftJoin(
   Join( Bgp(?x ex:teaches ex:sw),
       Join(Bgp(?x ex:age ?age),Bgp(?x rdf:type uni:Professor)),
   Bgp(?x ex:name ?name),
   true)
 FILTER (?age < 50) }
```

Transformation by example

- Apply Filter(.) for conditions

```
{Filter(?age < 50,
 LeftJoin(
   Join( Bgp(?x ex:teaches ex:sw),
        Join(Bgp(?x ex:age ?age),Bgp(?x rdf:type uni:Professor)),
   Bgp(?x ex:name ?name))
 )}
```

- In the end project the desired variables

```
Project((?x,?name),
 Filter(?age < 50,
   LeftJoin(
   Join( Bgp(?x ex:teaches ex:sw),
        Join(Bgp(?x ex:age ?age),Bgp(?x rdf:type uni:Professor)),
   Bgp(?x ex:name ?name))
```

Remainder of transformation

- The remainder of the transformation is easy to grasp
  - Just do similarly for UNIONs (or MINUS)
  - Formally the transformation is defined by a function *translate* that given a SPARQL query returns the SPARQL algebra expression
    - *translate* recursively applies over the structure of the SPARQL query
- You can check in the SPARQL Algebra specification at W3C
  - And test and validate in
    - http://sparql.org/query- validator.html

SPARQL operators

- *Bgp(P)*
  - match/evaluate *P*
- *Join(M_1,M_2)*
  - conjunctive join of the solution M_1 with solutions M_2
- *Union(M_1,M_2)*
  - union of the solutions M_1 with solutions M_2
- *LeftJoin(M_1,M_2)*
  - outer join of M_1 and M_2
- *Filter(M,C)*
  - select solutions in *M* that satisfy condition *C*
- *Minus(M1,M2)*
  - *The solutions of M1 which are not compatible with M2*

SPARQL evaluation

- Only *Bgp(.)* operates over triple patterns
- All other operators operate over **solutions**
- A solution is a partial function that:
  - given a (relevant) variable
  - returns a URI, a blank node or a Literal
- A **result** is a sequence (or list) of solutions

Solutions of Bgp(.)

- A partial function μ is a solution for $Bgp(P)$ over a graph G iff.
  - the domain of μ is exactly the set of variable in P
  - The exists an assignment σ of bnodes in P to URIs, bnodes or literals, such that $\mu(\sigma(P))$ is a subgraph of G
- The result of evaluating $Bgp(P)$ over graph G, written $[\![Bgp(P)]\!]_G$, is the multiset (i.e. a set, possibly with repetitions) of all solutions for $Bgp(P)$ over G
  - We denote a multiset S by a set of pairs (n,E), where (n,E) denotes that has n occurrence of E in S (in this case we say $M(E) = n$)
  - E.g. {(1,a),(2,b)} represents the multiset {a,b,b}, where M(a) = 1 and M(b) = 2
- We start by not considering the multiplicity of solutions

Bpg(.) example

```
ex:jja ex:teaches [ex:hasTopic "SPARQL"] .
ex:jja ex:teaches [ex:hasTopics "RDF"] .
```

- *Bgp*(`?x ex:teaches _:y . _:y ex:hasTopic ?t`) has 2 solutions
  - one assigning `ex:jja` to `?x` and `"SPARQL"` to `?t`
  - another, `ex:jja` to `?x` and `"RDF"` to `?t`
- *Bgp*(`?x ex:teaches _:y . _:y ex:hasTopic _:z`) has 1 solution with multiplicity 2
  - assigning `ex:jja` to `?x`
  - but with different replacements for `_:y` and `_:z`

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

Union of solutions

- Two solutions are compatible if for a same variable they both assign a same value
  - E.g. $x \mapsto a$, $y \mapsto b$ is not compatible with $x \mapsto c$
- The union of 2 compatible solutions μ_1 and μ_2 is $\mu_1 \cup \mu_2$ such that:
  - $\mu_1 \cup \mu_2(x) = \mu_1(x)$ if $x \in \mathrm{dom}(\mu_1)$
  - $\mu_1 \cup \mu_2(x) = \mu_2(x)$ otherwise
  - I.e. the union of matching assignments

Operations on sets of solutions

- $S_1 \bowtie S_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in S_1, \mu_2 \in S_2$, and μ_1 is compatible with $\mu_2\}$
- $S_1 \cup S_2 = \{\mu \mid \mu \in S_1$ or $\mu_2 \in S_2\}$
- $S_1 ⟕ S_2 = (S_1 \bowtie S_2) \cup (S_1 \setminus S_2)$, where

  $S_1 \setminus S_2 = \{\mu_1 \mid \mu_1 \in S_1$ and $\neg\exists \mu_2 \in S_2 \mid \mu_1$ and μ_2 are compatible$\}$

  i.e.

  $S_1 \setminus S_2 = \{\mu_1 \mid \mu_1 \in S_1$ and $\forall \mu_2 \in S_2 \mid \mu_1$ and μ_2 are incompatible$\}$

- $S_1 - S_2 = \{\mu_1 \mid \mu_1 \in S_1$ and $\forall \mu_2 \in S_2$, μ_1 and μ_2 are incompatible or $dom(\mu_1) \cap dom(\mu_2) = \emptyset\}$

Negation

- It was introduced in SPARQL 1.1 and provides two mechanisms:
  - Filtering results checking for the absence of a graph pattern (NOT EXISTS)
  - Removal of solutions with respect to other pattern (MINUS)

- The semantics of SPARQL has some subtleties which results in differences with respect to SQL.

EXISTS and MINUS

- The NOT EXISTS filter expression tests whether a graph pattern does not match the dataset, given the values of variables in the group graph pattern in which the filter occurs. It does not generate any additional bindings.

- EXISTS tests whether the pattern can be found in the data; it does not generate any additional bindings.

- MINUS evaluates both its arguments, then calculates solutions in the left-hand side that are not compatible with the solutions on the right-hand side.

Evaluation of expression

- $[\![Join(M_1,M_2)]\!]_G = [\![M_1]\!]_G \bowtie [\![M_2]\!]_G$
- $[\![Union(M_1,M_2)]\!]_G = [\![M_1]\!]_G \cup [\![M_2]\!]_G$
- $[\![LeftJoin(M_1,M_2)]\!]_G = [\![M_1]\!]_G ⟕ [\![M_2]\!]_G$
- $[\![Minus(M_1,M_2)]\!]_G = [\![M_1]\!]_G - [\![M_2]\!]_G$
- $[\![Filter(M,C)]\!]_G = \{\mu \in [\![M]\!]_G \mid \mu \vDash C\}$
  - where $\mu \vDash C$ means that μ makes condition C true.

Example queries

Data:

```
    [ :name "Paul" ; :phone "777-3426" ] .
    [ :name "John" ; :email "john@acd.edu" ] .
    [ :name "George" ; :webPage <www.george.edu> ] .
   [ :name "Ringo" ; :email "ringo@acd.edu" ;
     :webPage <www.starr.edu> ; :phone "888-4537" ] .
```

Patterns:

P1 = *Join*(*Bgp*(?A,:email,?E) , *Bgp*(?A,:webPage,?W))
P2 = *LeftJoin*(*Bgp*(?A,:email,?E), *Bgp*(?A,:webPage,?W))

```
P3 = LeftJoin(
          LeftJoin( Bgp(?A,:name,?N), Bgp(?A,:email,?E) ),
          Bgp(?A,:webPage,?W)
     )
```

Example queries

P4=*LeftJoin*(Bgp(?A,:name,?N),
LeftJoin(Bgp(?A,:email,?E),*Bgp*(?A,:webPage,?W))
)

P5 = *Join*(*Bgp*(?A,:name,?N),
Union(*Bgp*(?A,:email,?E), *Bgp* (?A,:webPage,?W))
)

P6 = *Filter*(*Bgp*(?A,:name,?N) *LeftJoin* *Bgp*(?A,:phone,?P) ,
?N = "Paul")

Multiplicity of solutions

- We have not considered the multiplicity of solutions (i.e. number of repeated solutions)
- For example, in the *Union*:
    - Let μ be a member of $[\![Union(M_1,M_2)]\!]_G$
    - Let $M_i(\mu)$ be the multiplicity of μ at $[\![M_i]\!]_G$
        - (or 0 if $\mu \notin [\![M_i]\!]_G$)
    - $M(\mu)$ in $[\![Union(M_1,M_2)]\!]_G$ is $M_1(\mu) + M_2(\mu)$
- For the remainder operators you can check the specification

Evaluation of SELECT

- The evaluation of a SPARQL query `SELECT` W `WHERE` P over a graph G is $\{\mu_{|W} \mid \mu \in [\![P]\!]_G\}$
  - $\mu_{|W}$ denotes the restriction of solutions in μ to variables in W
- Complexity results for the problem *"is μ a solution of query P over graph G?"*
  - If P only contains *Join* and *Filter*, the evaluation can be done in polynomial time
  - If P contains *Join* and *Union*, then the evaluation is NP-Complete
  - If P contains *LeftJoin*, then the evaluation is PSPACE-complete

Solution Modifiers

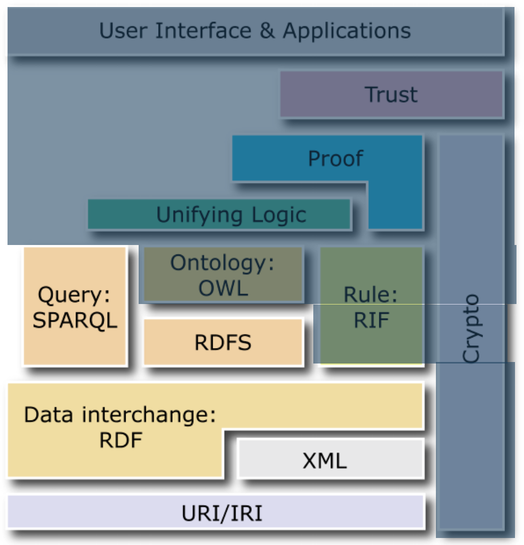
- The semantics of solution modifiers is given by additional operators
  - *ToList(M)*
    - Given a multiset *M* turns it into a list
  - *OrderBy(L,Comp)*
    - Sort the list *L* with the order specified in *Comp*
    - Together with *ToList(.)* is used for `ORDER BY`
  - *Distinct(M)*
    - Removes the duplicates in *M* (turns multiset *M* into a set)
    - To be used for `DISTINCT`
  - *Project(M,Vars)*
    - projects the variables in *Vars* in multiset *M*
  - *Slice(L,p,l)*
    - Cuts the solutions in list *L*, to a list of length *l*, starting from position *p*

Entailment regimes

- The definition of Bgp(.) only accounts for simple entailment
  - Subgraph matching
  - Only Bgp(.) generates solutions
- To answer queries according to other, stronger, semantics, instead of subgraph matching, something else is needed (***Entailment Regime***)
  - E.g. for RDFS (and other, such as OWL, later)
- Very naive solution:
  - First close the RDFS graph, and then use simple entailment
  - Doesn't work in practice!
- We will come back to this after studying OWL.

SPARQL Protocol

Querying the Web, in practice



SPARQL Protocol

- Besides the language and its semantics, SPARQL comes with a protocol, and service descriptions, to put it to work
- It specifies how queries can be sent to endpoints in the Web, and how results (and errors) are returned
- Query:
  - `GET`: Query is part of the URL
    - `http://server…/endpoint?query=…`
  - `POST`: Query is in the body of the HTTP request, e.g. via an HTML form
- Update
  - POST with content-type application/sparql update or via HTML form

Service Descriptions

- Define method and vocabulary for describing SPARQL endpoints
- Client/User can request information about the SPARQL service:
  - supported extension functions
  - used data set
  - supported entailment regimes
  - …
- See full specification at
  - `http://www.w3.org/TR/sparql11-service-description/`

HTTP trace example

- Request

```
GET /sparql/ HTTP/1.1
Host: www.example
Accept: text/turtle
```

- Response

```
HTTP/1.1 200 OK
Date: Fri, 09 Oct 2009 17:31:12 GMT
Server: Apache/1.3.29 (Unix) PHP/4.3.4 DAV/1.0.3
Connection: close
Content-Type: text/turtle
@prefix sd: <http://www.w3.org/ns/sparql-service-description#> .
…
[] a sd:Service ;
    sd:endpoint <http://www.example/sparql/> ;
    sd:supportedLanguage sd:SPARQL11Query ;
    sd:resultFormat format:RDF_XML, <http://www.w3.org/ns/formats/Turtle> ;
    sd:feature sd:DereferencesURIs ;
    sd:defaultEntailmentRegime ent:RDFS ;
    sd:defaultDataset [a sd:Dataset ; sd:defaultGraph
                                 [a sd:Graph ; void:triples 100] ;
     sd:namedGraph [a sd:NamedGraph ;
            sd:name <http://www.example/named-graph> ;
            sd:entailmentRegime ent:OWL-RDF-Based ;
            sd:supportedEntailmentProfile prof:RL ;
            sd:graph [a sd:Graph ; void:triples 2000]
…
```

Federated queries

- Other SPARQL endpoints can be queried inside a SPARQL query with SERVICE construct. E.g.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?person ?interest ?known
WHERE
{
  SERVICE <http://people.example.org/sparql>
    { ?person foaf:name ?name .
    OPTIONAL {
      ?person foaf:interest ?interest .
      SERVICE <http://people2.example.org/sparql>
        { ?person foaf:knows ?known . } }
  }
}
```

SPARQL in Jena

- SPARQL server with Jena Fuseki
- SPARQL functionality in Jena
  - http://jena.apache.org/documentation/javadoc/arq
- Main classes in com.hp.hpl.jena.query
  - Query in SPARQL
  - QueryFactory for creating queries
  - QueryExecution for the execution state of a query
  - QueryExecutionFactory for creating query executions
  - ResultSet for results of a SELECT query

SPARQL example in JENA

```
String qStr = "SELECT ?a ?b ...";
Query q = QueryFactory.create(qStr);
QueryExecution qe =
          QueryExecutionFactory.create(q,model);
try {
          res = qe.execSelect();
          while( res.hasNext()) {
                    QuerySolution soln = res.next();
                    RDFNode a = soln.get("?a");
                    RDFNode b = soln.get("?b");
                    System.out.println(""+a+" knows "+b);
}
} finally {
          qe.close();
}
```

Remote SPARQL with Jena

- Jena can also be used to send SPARQL queries to remote endpoints. E.g.

```
String endpoint = "http://dblp.l3s.de/d2r/sparql";
String qStr = "SELECT ?a ?b ...";
Query q = QueryFactory.create(qStr);
QueryExecution qe =
   QueryExecutionFactory.sparqlService(endpoint,q);
try {
        res = qe.execSelect();
        ...
} finally {
qe.close();
}
```

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA