An Introduction to SPARQL

V. CHRISTOPHIDES

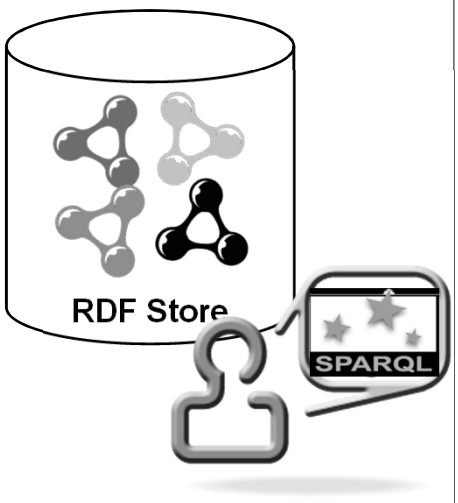
Department of Computer Science
University of Crete
ICS - FORTH, Heraklion, Crete

Introduction

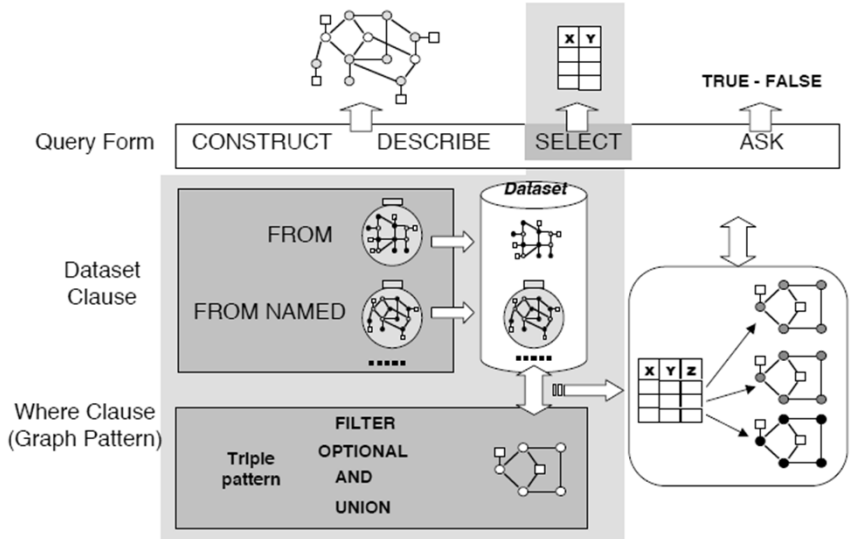
- SPARQL is the W3C candidate recommendation graph-matching query language for RDF
  - The acronym stands for SPARQL Protocol and RDF Query Language
- A SPARQL query consists of three parts:
  - Pattern matching: optional, union, nesting, filtering.
  - Solution modifiers: projection, distinct, order, limit, offset.
  - Output part: construction of new triples, …
- In addition to the core language, W3C has also defined:
  - The SPARQL Protocol for RDF specification: it defines the remote protocol for issuing SPARQL queries and receiving the results
  - The SPARQL Query Results XML Format specification: it defines an XML document format for representing the results of SPARQL queries
  - The SPARQL Federated Query: it extends SPARQL for executing queries distributed over different SPARQL endpoints
  - The SPARQL Service Description: it is a method for discovering and a vocabulary for describing SPARQL services

What are the Design Challenges in SPARQL?

- SPARQL has to take into account the distinctive features of RDF:
  - Should be able to extract information from interconnected RDF graphs
  - Should be consistent with the open-world semantics of RDF
    - Should offer the possibility of adding optional information if present
  - Should be able to properly interpret RDF graphs with a vocabulary with predefined semantics
  - Should offer some functionalities for navigating in an RDF graph



SPARQL in a Nutshell



C. Gutierrez – Foundations of RDF Databases - ESWC 2008

SPARQL Query Forms

- The `SELECT` query form returns variable bindings
- The `CONSTRUCT` query form returns an RDF graph specified by a graph template
- The `ASK` query form can be used to test whether or not a graph pattern has a solution
  - No information is returned about the possible query solutions, just whether or not a solution exists
- There is also a `DESCRIBE` query form which is not important and SPARQL does not prescribe any semantics for it

Solution Sequences and Modifiers

- Graph patterns in a WHERE clause generate an unordered collection of solutions, each solution being a mapping i.e.,
  - a partial function from variables to RDF terms (URIs, Literals, Bnodes), aka variable binding
- These solutions are then treated as a sequence initially in no specific order which is used to generate the results of a SPARQL query
- A solution sequence modifier can be applied to create another sequence:
  - Order modifier: put the solutions in some given order
  - Projection modifier: choose certain variables. This is done using the SELECT clause
  - Distinct modifier: ensure solutions in the sequence are unique
  - Reduced modifier: permit elimination of some non-unique solutions
  - Offset modifier: control where the solutions start from, in the overall sequence of solutions
  - Limit modifier: restrict the number of solutions

SPARQL Graph Patterns

- To define graph patterns, we must first define triple patterns:
  - A triple pattern is like an RDF triple, but with the option of a variable in place of RDF terms (i.e., IRIs, literals or blank nodes) in the subject, predicate or object positions
  - Example: `<http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> ?title .`
    `?title` is a variable
- We can distinguish the following kinds of graph patterns:
  - Group graph patterns are the more general case of patterns build out of:
    - Basic graph patterns
    - Optional graph patterns
    - Alternative graph patterns
    - Filter conditions
  - Patterns on named graphs

Basic Graph Patterns

- A basic graph pattern (BGP) is a set of triple patterns written as a sequence of triple patterns (separated by a period if necessary)
  - A BGP should be understood as the conjunction of its triple patterns
- **Example**:

```
?x foaf:name ?name . ?x foaf:mbox ?mbox
```

- **Note**: There is no keyword for conjunction (e.g., AND) in SPARQL
  - Conjunctive triple patterns or BGPs are simply juxtaposed and then enclosed in { and } to form a group graph pattern

Group Graph Patterns

- A group graph pattern is a set of graph patterns delimited with braces { }
  - { P1 P2 }
  - {} is the empty group graph pattern

- Simple examples:

```
{ ?x foaf:name ?name . ?x foaf:mbox ?mbox }

{ ?x foaf:name ?name . ?x foaf:mbox ?mbox . }

{ { ?x foaf:name ?name . } { ?x foaf:mbox ?mbox . } }
```

- The above three group graph patterns are equivalent:
  - When a group graph pattern consists only of triple patterns or only of BGPs, these patterns are interpreted conjunctively, and the group graph pattern is equivalent to the corresponding set of triple patterns

GGP Example

- **Data**:

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

- **Query**:

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

- **Result**:

?name	?mbox
"Peter Goodguy"	<mailto:peter@example.org>
"Johnny Lee Outlaw"	<mailto:jlow@example.com>

Optional Graph Patterns (Outer-Joins)

- Regular, complete structures cannot be assumed in all RDF graphs
- It is useful to have queries that allow information to be added to the answer where the information is available, but do not reject the answer because some part of the query pattern does not match
  - Optional graph pattern matching provides this facility: if the optional part does not match, it creates no bindings but does not eliminate the solution
- Optional parts of a complex graph pattern that we are trying to compute may be specified by starting with a graph pattern `P1` and then applying the keyword `OPTIONAL` to another graph pattern `P2` that follows it:
  - `{P1 OPTIONAL { P2 }}`
  - The group graph pattern following a keyword OPTIONAL can of course be as complex as possible e.g., it can contain a FILTER

OGP Example

- Data:

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

- Query:

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

- Result:

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

Alternative Patterns (Disjunction)

- SPARQL provides a means of forming the disjunction of graph patterns with the keyword `UNION` so that one of several alternative graph patterns may match
  - If more than one of the alternatives match, all the possible pattern solutions are found
- Alternative graph patterns that are combined by UNION are processed independently of each other and the results are combined using (set-theoretic) union
  - `{P1} UNION {P2}`

AGP Example

- **Data:**

```
@prefix dc10: <http://purl.org/dc/elements/1.0/> .
@prefix dc11: <http://purl.org/dc/elements/1.1/> .
_:a dc10:title "SPARQL Query Language Tutorial" .
_:a dc10:creator "Alice" .
_:b dc11:title "SPARQL Protocol Tutorial" .
_:b dc11:creator "Bob" .
_:c dc10:title "SPARQL" .
_:c dc11:title "SPARQL (updated)" .
```

- **Query:**

```
PREFIX dc10: <http://purl.org/dc/elements/1.0/>
PREFIX dc11: <http://purl.org/dc/elements/1.1/>
SELECT ?title
WHERE {{?book dc10:title ?title
       UNION
       {?book dc11:title ?title
      }
```

- **Result:**

?title
"SPARQL Protocol Tutorial"
"SPARQL"
"SPARQL (updated)"
"SPARQL Query Language Tutorial"

AGP Example (cont'd)

- **Query:**

```
PREFIX dc10: <http://purl.org/dc/elements/1.0/>
PREFIX dc11: <http://purl.org/dc/elements/1.1/>
SELECT ?author ?title
WHERE {{?book dc10:title ?title. ?book dc10:creator ?author.}
       UNION
       {?book dc11:title ?title. ?book dc11:creator ?author.}
      }
```

- **Result:**

?author	?title
"Bob"	"SPARQL Protocol Tutorial"
"Alice"	"SPARQL Query Language Tutorial"

Queries with RDF Literals

- We have to be careful when matching RDF literals
- **Data**:

```
@prefix dt: <http://example.org/datatype#> .
@prefix ns: <http://example.org/ns#> .
@prefix : <http://example.org/ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
:x ns:p "cat"@en .
:y ns:p "42"^^xsd:integer .
:z ns:p "abc"^^dt:specialDatatype .
```

- The queries `SELECT ?v WHERE { ?v ?p "cat" }` and `SELECT ?v WHERE { ?v ?p "cat"@en }` have different results
- Only the second one finds a matching triple and returns:

?v
<http://example.org/ns#x>

Filter Expressions

- The FILTER construct restricts variable bindings to those for which the filter expression evaluates to TRUE by considering
  - equality = among variables and RDF terms
  - unary predicate `bound`
  - Boolean combinations (∧, ∨, ¬)
- Group graph patterns are used to restrict the scope of FILTER conditions
- A FILTER condition is a restriction on solutions over the whole group in which the filter appears
  - P_1 . P_2 . FILTER (... boolean expression...)
- We can have multiple FILTERs in a group graph pattern
  - They are equivalent to a single filter with conjoined filter conditions
- FILTERs can be very complex Boolean conditions !
  - The regular expression language used by `regex` is defined in XQuery 1.0 and XPath 2.0

Constraints on Variables

- **Data**:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix : <http://example.org/book/> .
@prefix ns: <http://example.org/ns#> .
:book1 dc:title "SPARQL Tutorial" .
:book1 ns:price 42 .
:book2 dc:title "The Semantic Web" .
:book2 ns:price 23 .
```

- **Query**:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
SELECT ?title ?price
WHERE { ?x ns:price ?price .
FILTER (?price < 30.5)
?x dc:title ?title . }
```

- **Result**:

?title	?price
"The Semantic Web"	23

Blank Nodes in Query Results

- **Data**:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:b foaf:name "Bob" .
```

- **Query**:

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

- **Result**:

?x	?name
_:c	"Alice"
_:d	"Bob"

Blank Nodes in Query Results (cont'd)

- **Data**:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:b foaf:name "Bob" .
_:a foaf:knows _:b .
_:b foaf:knows _:a .
```

- **Query**:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x ?name1 ?y ?name2
WHERE { ?x foaf:name ?name1 . ?y foaf:name ?name2 .
?x foaf:knows ?y }
```

- **Result**:

?x	?name	?y	?name
_:c	"Alice"	_:d	"Bob"
_:d	"Bob"	_:c	"Alice"

Blank Nodes in Graph Patterns

- Blank nodes in graph patterns act as variables, not as references to specific blank nodes in the data being queried
- Blank nodes cannot appear in a SELECT clause
- The scope of blank node is the BGP in which it appears
  - A blank node which appears more than once in the same BGP stands for the same RDF term
- The same blank node is not allowed to appear in two BGPs of the same query
- **Important**: there is no reason to use blank nodes in a query; you can get the same functionality using variables

Blank Nodes in Graph Patterns Example

- **Data**:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:b foaf:name "Bob" .
_:a foaf:knows _:b .
_:b foaf:knows _:a .
```

- **Query**:

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

- **Result**:

?name
"Alice"
"Bob"

Blank Nodes in Graph Patterns Example

- **Data**:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:b foaf:name "Bob" .
_:a foaf:knows _:b .
_:b foaf:knows _:a .
```

- **Query**:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name1 ?name2
WHERE { _:z foaf:name ?name1 . _:v foaf:name ?name2 .
        _:z foaf:knows _:v }
```

- **Result**:

?name1	?name2
"Alice"	"Bob"
"Bob"	"Alice"

A Complete SPARQL Example

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX oo: <http://purl.org/openorg/>                    PREFIX DECLARATIONS
SELECT ?name ?expertise                                  RESULT CLAUSE
FROM <http://data.uoc.gr/>                               DATASET CLAUSE
WHERE {
 ?person foaf:name ?name .   ; foaf:familyName ?surname .
 ?person rdf:type foaf:Person .
 ?person foaf:title ?title . FILTER regex(?title, "^Prof")
 OPTIONAL {
  ?person oo:availableToCommentOn ?expertiseURI .
  ?expertiseURI rdfs:label ?expertise
 }
}                                                        QUERY CLAUSE
ORDER BY ?surname                                        SOLUTION MODIFIERS
```

Give me a list of names of professors in University of Crete and their expertise (if available), in order of their surname

SPARQL in a Nutshell

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX oo: <http://purl.org/openorg/>                    PREFIX DECLARATIONS
SELECT ?name ?expertise                                  RESULT CLAUSE
FROM <http://data.uoc.gr/>                               DATASET CLAUSE
WHERE {
 ?person foaf:name ?name ; foaf:familyName ?surname .
 ?person rdf:type foaf:Person .
 ?person foaf:title ?title . FILTER regex(?title, "^Prof")
 OPTIONAL {
  ?person oo:availableToCommentOn ?expertiseURI .
  ?expertiseURI rdfs:label ?expertise
 }
}                                                        QUERY CLAUSE
ORDER BY ?surname                                        SOLUTION MODIFIERS
```

Shortcuts for URIs

Which results do you want?

Where should we look?

What are you looking for?

How should results be ordered/split?

Give me a list of names of professors in University of Crete and their expertise (if available), in order of their surname

A Complete SPARQL Example

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX oo: <http://purl.org/openorg/>                    PREFIX DECLARATIONS
SELECT ?name ?expertise                                  RESULT CLAUSE
FROM <http://data.uoc.gr/>                               DATASET CLAUSE
WHERE {
 ?person foaf:name ?name ; foaf:familyName ?surname .
 ?person rdf:type foaf:Person .
 ?person foaf:title ?title . FILTER regex(?title, "^Prof")
 OPTIONAL {
  ?person oo:availableToCommentOn ?expertiseURI .
  ?expertiseURI rdfs:label ?expertise
 }
}                                                        QUERY CLAUSE
ORDER BY ?surname                                        SOLUTION MODIFIERS
```

Give me a list of names of professors in University of Crete and their expertise (if available), in order of their surname

Prefix Declarations

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX oo: <http://purl.org/openorg/>
```

PREFIX DECLARATIONS

foaf:Person ⇔ <http://xmlns.com/foaf/0.1/Person>

Use http://prefix.cc/ …

A Complete SPARQL Example

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX oo: <http://purl.org/openorg/>
```

PREFIX DECLARATIONS

```
SELECT ?name ?expertise
```

RESULT CLAUSE

```
FROM <http://data.uoc.gr/>
```

DATASET CLAUSE

```
WHERE {
 ?person foaf:name ?name ; foaf:familyName ?surname .
 ?person rdf:type foaf:Person .
 ?person foaf:title ?title . FILTER regex(?title, "^Prof")
 OPTIONAL {
  ?person oo:availableToCommentOn ?expertiseURI .
  ?expertiseURI rdfs:label ?expertise
 }
}
```

QUERY CLAUSE

```
ORDER BY ?surname
```

SOLUTION MODIFIERS

Give me a list of names of professors in University of Crete and their expertise (if available), in order of their surname

Solution Modifiers

```
ORDER BY ?surname                                    SOLUTION MODIFIERS
```

Order output results by surname (as you probably guessed)

…also…

LIMIT

Only return 10 results

```
ORDER BY ?surname LIMIT 10                           SOLUTION MODIFIERS
```

OFFSET

Return results 20–30

```
ORDER BY ?surname LIMIT 10 OFFSET 20                 SOLUTION MODIFIERS
```

Give me a list of names of professors in University of Crete and their expertise (if available), in order of their surname

OFFSET and LIMIT clauses

- The `LIMIT` clause puts an upper bound on the number of solutions returned
  - If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned
- The `OFFSET` clause causes the solutions generated to start after the specified number of solutions
  - An `OFFSET` of zero has no effect
- Using `LIMIT` and `OFFSET` to select different subsets of the query solutions is not useful unless the order is made predictable by using `ORDER BY`

Removing Duplicates

- By default, SPARQL query results may contain duplicates (so the result of a SPARQL query is a bag not a set)
- The modifier `DISTINCT` enforces that no duplicates are included in the query results
- The modifier `REDUCED` permits the elimination of duplicates (the implementation decides what to do e.g., based on optimization issues)

Building Blocks for SPARQL Semantics

- Given an RDF graph the evaluation of a triple pattern t is the set of mappings that
  - ◆make t to match the graph
  - ◆have as domain the variables in t
- Two mappings are compatible if they agree in their shared variables

	?X	?Y	?Z	?V
μ_1 :	R_1	john		
μ_2 :	R_1		J@edu.ex	
μ_3 :			P@edu.ex	R_2
$\mu_1 \cup \mu_2$:	R_1	john	J@edu.ex	
$\mu_1 \cup \mu_3$:	R_1	john	P@edu.ex	R_2

  μ_2 and μ_3 are not compatible ()
- A mapping satisfies
  - ◆ `?X = c` if it gives the value `c` to variable `?X`
  - ◆ `?X =?Y` if it gives the same value to `?X` and `?Y`
  - ◆ `bound(?X)` if it is defined for `?X`

SPARQL Algebraic Operators

- SPARQL algebra defines 5 operators on mapping bags
  - Unary ops:
    - π (projection),
    - σ (selection, also called filtering)
  - Binary ops:
    - U (union)
    - $\bowtie$ (join)
    - $\supset\!\bowtie$ (optional)

SPARQL Algebraic Operators

- SPARQL algebra defines 5 operators on mapping bags
  - Unary ops:
    - π (projection),
    - σ (selection, also called filtering)
  - Binary ops:
    - U (union)
    - $\bowtie$ (join)
    - $\supset\!\bowtie$ (optional)

Ω

?x	?y
a	b
a	c
d	e

$\pi_{?x}$ (Ω)

?x
a
d

μ_1 **card(μ_1) = 2**

μ_2 **card(μ_2) = 1**

SPARQL Algebraic Operators

- SPARQL algebra defines 5 operators on mapping bags
  - Unary ops:
    - π (projection),
    - σ (selection, also called filtering)
  - Binary ops:
    - U (union)
    - $\bowtie$ (join)
    - $\mathbin{⟕}$ (optional)

Ω

?x	?y
a	b
a	c
d	e

$\sigma_{?x=a}(\Omega)$

?x	?y
a	b
a	c

μ_1 **card(μ_1) = 1**

μ_2 **card(μ_2) = 1**

SPARQL Algebraic Operators

- SPARQL algebra defines 5 operators on mapping bags
  - Unary ops:
    - π (projection),
    - σ (selection, also called filtering)
  - Binary ops:
    - U (union)
    - $\bowtie$ (join)
    - $\mathbin{⟕}$ (optional)

Ω_1

?x	?y
a	b

μ_1

Ω_2

?x	?z
c	d

μ_2

$\Omega_1 \cup \Omega_2$

?x	?y	?z	
a	b	-	μ_1
c	-	d	μ_2

?z is unbound in μ_1

SPARQL Algebraic Operators

- SPARQL algebra defines 5 operators on mapping bags
  - ◆Unary ops:
    - π (projection),
    - σ (selection, also called filtering)
  - ◆Binary ops:
    - U (union)
    - $\bowtie$ (join)
    - $\mathbin{⟕}$ (optional)

μ and μ' are compatible (μ ~ μ'), if they agree on their common variables

$\mu_1 \sim \mu_4$, $\mu_3 \sim \mu_4$

$\mu_2 \not\sim \mu_4$

Ω_1

?x	?y
a	b
c	d
e	-

μ_1
μ_2
μ_3

Ω_2

?y	?z
b	f

μ_4

$\Omega_1 \bowtie \Omega_2$

?x	?y	?z
a	b	f
e	b	f

$\mu_5 = \mu_1 \cup \mu_4$

$\mu_6 = \mu_3 \cup \mu_4$

SPARQL Algebraic Operators

- SPARQL algebra defines 5 operators on mapping bags
  - ◆Unary ops:
    - π (projection),
    - σ (selection, also called filtering)
  - ◆Binary ops:
    - U (union)
    - $\bowtie$ (join)
    - $\mathbin{⟕}$ (optional)

(P1 OPTIONAL P2) can be written as
(P1 AND P2) UNION (P1 MINUS P2)

(a, b, f): $\Omega_1 \bowtie \Omega_2$

(c, d, -): $\Omega_1 \setminus \Omega_2$

$\Omega_1 \mathbin{⟕} \Omega_2$

Ω_1

?x	?y
a	b
c	d

μ_1
μ_2

Ω_2

?y	?z
b	f

μ_3

?x	?y	?z
a	b	f
c	d	-

$\mu_4 = \mu_1 \cup \mu_3$

μ_2

Few SPARQL Algebraic Equivalences

$$\sigma_{?x=?x}(P_1) \equiv P_1 \qquad P_1 \bowtie \emptyset \equiv \emptyset \qquad P_1 \cup \emptyset \equiv P_1$$
$$P_1 \cup P_2 \equiv P_2 \cup P_1 \qquad P_1 \bowtie P_2 \equiv P_2 \bowtie P_1$$
$$P_1 \cup (P_2 \cup P_3) \equiv (P_1 \cup P_2) \cup P_3$$
$$P_1 \bowtie (P_2 \bowtie P_3) \equiv (P_1 \bowtie P_2) \bowtie P_3$$
$$P_1 \bowtie (P_2 \cup P_3) \equiv (P_1 \bowtie P_2) \cup (P_1 \bowtie P_3)$$

$$P_1 \setminus P_1 \equiv \emptyset \qquad P_1 \setminus (P_2 \cup P_3) \equiv (P_1 \setminus P_2) \setminus P_3$$
$$P_1 \bowtie (P_2 \setminus P_3) \equiv (P_1 \bowtie P_2) \setminus P_3$$

$$(P_1 \setminus (P_1 \setminus P_2)) \cup (P_1 \setminus P_2) \equiv P_1$$

39

Negation in SPARQL

- SPARQL offers two forms of negation:
  - The Boolean "not" (`!`) operator in FILTER conditions
  - A limited form of negation as failure which can be simulated using `OPTIONAL`, `FILTER` and `!bound`
- SPARQL 1.0 does not offer an explicit algebraic difference operator but this operator is implicit in the definition of the `OPTIONAL` operator
  - SPARQL 1.1. offers more non-monotonic operators

40

Negation in FILTER conditions

- **Data:**

```
@prefix ns: <http://example.org/ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
_:a ns:p "42"^^xsd:integer .
```

- **Query:**

```
PREFIX ns: <http://example.org/ns#>
SELECT ?v
WHERE {
        ?v ns:p ?y . FILTER (?y != 42)
      }
```

- **Result:**

?v

The Operator bound

- The expression `bound(var)` is one of the expressions allowed in FILTER conditions
- Given a mapping to which FILTER is applied, `bound(var)` evaluates to true if `var` is bound to a value in that mapping and false otherwise

Example

- **Data:**

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.org/schema/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
_:a foaf:givenName "Alice“ .
_:b foaf:givenName "Bob" .
_:b ex:age “30"^^xsd:integer .
_:m foaf:givenName “Mike" .
_:m ex:age “65"^^xsd:integer .
```

- **Query:** Find the names of people with name and age

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/schema/>
SELECT ?name
WHERE { ?x foaf:givenName ?name . ?x ex:age ?age }
```

- **Result:**

?name
“Bob"
“Mike"

Examples with Negation

- **Query**: Find people with a name but **no** expressed age:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/schema/>
SELECT ?name
WHERE { ?x foaf:givenName ?name .
        OPTIONAL { ?x ex:age ?age }
        FILTER (!bound(?age))
      }
```

- **Result**:

? name
“Alice"

Examples with Negation (cont'd)

- **Query:** Find the names of people with name but **no** expressed age **or** age less than 60 years

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/schema/>
SELECT ?name
WHERE { ?x foaf:givenName ?name .
        OPTIONAL { ?x ex:age ?age .
                   FILTER(?age >= 60) } .
        FILTER (!bound(?age))
      }
```

- **Result:**

?name
"Alice"
"Bob"

Examples with Negation (cont'd)

- Note that the OPTIONAL pattern in the previous query does not generate bindings in the following two cases:
  - There is no `ex:age` property for ?x (e.g., when ?x=_a)
  - There is an `ex:age` property for ?x but its value is less than 60 (e.g., when ?x=_b)
- These two cases are then selected for output by the FILTER condition that uses `!bound`
- In the previous examples where we used

```
{ P1 OPTIONAL P2 } FILTER(!bound(?x) }
```

to express negation as failure, the variable ?x appeared in graph pattern P2 but not in graph pattern P1 otherwise we cannot have the desired effect
  - this simple idea might not work in more complicated cases

Closed World Assumption (CWA) and Negation as Failure (NF)

- We saw that it is possible to simulate a non-monotonic construct like negation as failure through SPARQL language constructs
- However, SPARQL makes no assumption to interpret statements in an RDF graph using negation as failure or some other non-monotonic assumption (e.g., Closed World Assumption)
  - SPARQL, but also RDF(S) make the Open World Assumption i.e. things that are not known to be true or false are assumed to be possible
- Monotonicity: Let *KB* be a set of FOL formulas and *φ* and *θ* two arbitrary FOL formulas
  - If *KB* entails *φ* then *KB* union { *θ* } entails *φ* as well
- CWA and NF result in non-monotonicity
  - If A is a ground atomic formula in FOL, then the CWA says:
    - If KB does not entail A, then assume not A to be entailed
  - If A is a ground atomic formula in FOL, then negation as failure says:
    - If you cannot prove A from the KB, then assume not A has been proven

47

OWA vs. CWA in RDF

- DB: `tall(John)`
- Query: `?-tall(John)`
  Answer: `yes`
- Query: `?-tall(Mike)`
  Answer: `no` (using the CWA or negation as failure)
- Update DB with `tall(Mike)`
- Query: `?-tall(Mike)`
  Answer: `yes`

- DB: `tall(John)`
- Query: `?-tall(John)`
  Answer: `yes`
- Query: `?-tall(Mike)`
  Answer: `I don't know` (using the OWA)

- In general, the OWA is the most natural assumption to make in RDF since we are writing incomplete Web resource descriptions and we expect that these resource descriptions will be extended and reused by us or others later on
- But even in the context of the Web, there are many examples where the CWA is more appropriate (e.g., when we describe the schedule of a course we give the times the course takes place; the course does not take place at any other time)
- It would be nice to have facilities to say what assumption to make in each case

48

New Features of SPARQL 1.1

- New query features:
  - Aggregate functions
  - Subqueries
  - Negation (explicit)
  - Expressions in the SELECT clause
  - Property Paths
  - Assignment
  - A short form for CONSTRUCT
  - An expanded set of functions and operators
- Updates
- Federated queries

Aggregates

- Aggregate functions can be used to do computations over groups of solutions that satisfy certain graph patterns
  - By default a solution set consists of a single group containing all solutions
  - The following functions are allowed: `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, `GROUP_CONCAT`, and `SAMPLE`
- Grouping is specified using the `GROUP BY` clause
- The `HAVING` clause can also be used to constrain grouped solutions in the same way `FILTER` constrains ungrouped ones

Example: Aggregates

- **Data:**

```
@prefix : <http://books.example/> .
:org1 :affiliates :auth1, :auth2 .
:auth1 :writesBook :book1, :book2 .
:book1 :price 9 .
:book2 :price 5 .
:auth2 :writesBook :book3 .
:book3 :price 7 .
:org2 :affiliates :auth3 .
:auth3 :writesBook :book4 .
:book4 :price 7 .
```


Example (cont'd)

- **Query:** Find the total price of books written by authors affiliated with some organization: output organization id and total price only if the total price is greater than 10

```
PREFIX : <http://books.example/>
SELECT (?org SUM(?lprice) AS ?totalPrice)
WHERE { ?org :affiliates ?auth .
        ?auth :writesBook ?book .
        ?book :price ?lprice . }
GROUP BY ?org
HAVING (SUM(?lprice) > 10)
```

- **Result:**

?org	?totalPrice
<http://books.example/org1>	21

Subqueries

- Subqueries are a way to embed SPARQL queries inside other queries to allow the expression of requests that are not possible otherwise
  - Subqueries are evaluated first and then the outer query is applied to their results
  - Only variables projected out of the subquery (i.e., appearing in its SELECT clause) will be visible to the outer query
- Consider graph patterns P_1 , P_2 and P_3 :

```
P1 .
{SELECT
 WHERE {P2}
}
P3 .
```

  - Join the results of the subquery with the results of solving P_1 and P_3
- Subqueries are useful when combining limits and aggregates with other constructs

Example: Subqueries

- **Data:**

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <http://sales.com/> .
:sale1 :company :c1; :amount 7500^^xsd:integer; :year "2011".
:sale2 :company :c1; :amount 17000^^xsd:integer;:year "2011".
:sale3 :company :c1; :amount 5500^^xsd:integer; :year "2012".
:sale4 :company :c1; :amount 7000^^xsd:integer; :year "2012".
:sale5 :company :c2; :amount 3000^^xsd:integer; :year "2011".
:sale6 :company :c2; :amount 4000^^xsd:integer; :year "2011".
:sale7 :company :c2; :amount 5000^^xsd:integer; :year "2012".
:sale8 :company :c2; :amount 6000^^xsd:integer; :year "2012".
```

Example (cont'd)

- **Query:** Find companies that increased their sales from 2011 to 2012 and the amount of increase

```
PREFIX : <http://sales.com/>
SELECT ?c ((?total2012 - ?total2011) AS ?increase)
WHERE {
       { SELECT ?c (SUM(?m) AS ?total2012)
         WHERE { ?s a :Sale ; :company ?c ;
                  :amount ?m ; :year: "2012" . }
         GROUP BY ?c  } .
       { SELECT ?c (SUM(?m) AS ?total2011)
         WHERE { ?s a :Sale ; :company ?c ;
                  :amount ?m ; :year: "2011" . }
         GROUP BY ?c  } .
       FILTER (?total2012 > ?total2011)
       }
```

- **Result:**

?c	?increase
<http://sales.com/c2>	"4000"^^<http://www.w3.org/2001/XMLSchema#integer>

Negation

- In SPARQL 1.1 we have two ways to express negation:
  - The algebraic operator MINUS (is the same as the difference operation of the SPARQL algebraic semantics) used for removing matches based on the evaluation of two query patterns
    - P_1 MINUS P_2 returns all the mappings in P_1 that are incompatible with all mappings in P_2 or they have disjoin domains (different vars)
  - The operator NOT EXISTS in FILTER expressions used for testing whether a pattern exists in the data, given the bindings already determined by the query pattern
    - P_1 NOT EXISTS P_2 returns true if the pattern P_2 do not matches the data after substituting all mappings in P_1 (shared vars)
- Proposition: `(P1 MINUS P2)` is equivalent to:
  `P1 OPT (P2 AND (?X1, ?X2, ?X3)) FILTER ¬bound(?X1)`,
  where `?X1, ?X2, ?X3` are mentioned neither in `P1` nor in `P2`
- Unlike MINUS, NOT EXISTS should only be in a FILTER expression

Example: MINUS

- **Data:**

```
@prefix : <http://example/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:alice foaf:givenName "Alice" ; foaf:familyName "Smith" .
:bob foaf:givenName "Bob" ; foaf:familyName "Jones" .
:carol foaf:givenName "Carol" ; foaf:familyName "Smith" .
```

- **Query:** Find all persons that do not have given name “Bob”

```
PREFIX : <http://example/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?s
WHERE { ?s ?p ?o .
        MINUS { ?s foaf:givenName "Bob" . } }
```

- **Result:**

?s
<http://example/carol>
<http://example/alice>

Example: NOT EXISTS

- **Data:**

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

- **Query:** Find persons for whom we have **no** name

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person
WHERE { ?person rdf:type foaf:Person .
        FILTER NOT EXISTS { ?person foaf:name ?name } }
```

- **Result:**

?person
<http://example/bob>

- This is what we expressed earlier with OPTIONAL, FILTER and !bound

NOT EXISTS vs MINUS

- MINUS and NOT EXISTS do not always return the same result if they are not applied with care
- Example:

```
PREFIX : <http://example/>
SELECT *
WHERE { ?s ?p ?o FILTER NOT EXISTS { :a :b :c } }
```

evaluates to an empty result set since there is no mapping set for P_2

```
PREFIX : <http://example/>
SELECT *
WHERE { ?s ?p ?o MINUS { :a :b :c } }
```

evaluates to result set with one query solution because there is no match of bindings and so no solutions are eliminated

s	p	o
<http://example/a>	<http://example/b>	<http://example/c>

NOT EXISTS vs MINUS

```
@prefix : <http://example/> .
:a :b :c .
SELECT *
WHERE { ?s ?p ?o FILTER NOT EXISTS { ?x ?y ?z } }
```

evaluates to an empty result set because `{ ?x ?y ?z }` matches given any `?s ?p ?o`, so `NOT EXISTS { ?x ?y ?z }` eliminates any solution

```
SELECT *
WHERE { ?s ?p ?o MINUS { ?x ?y ?z } }
```

since there is no shared variable between the first part `(?s ?p ?o)` and the second `(?x ?y ?z)` no bindings are eliminated, so the result is

s	p	o
<http://example/a>	<http://example/b>	<http://example/c>

See more examples
http://www.openlinksw.com/uda/wiki/main/Main/VirtTipsAndTricksGuideSPARQLNOTEXISTS

Example: Expressions in the SELECT clause

- **Data:**

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix : <http://example.org/book/> .
@prefix ns: <http://example.org/ns#> .
:book1 dc:title "SPARQL Tutorial" .
:book1 ns:price 42 .
:book1 ns:discount 0.2 .
:book2 dc:title "The Semantic Web" .
:book2 ns:price 23 .
:book2 ns:discount 0.25 .
```

- **Query:** Find all book titles and their discounted price

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
SELECT ?title (?p*(1-?discount) AS ?price)
WHERE { ?x ns:price ?p .
        ?x dc:title ?title .
        ?x ns:discount ?discount}
```

- **Result:**

?title	?price
"The Semantic Web"	17.25
"SPARQL Tutorial"	33.6

Example: Expressions in the SELECT clause (cont'd)

- **Query:** Find all book titles, their full price and their discounted price

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
SELECT ?title (?p AS ?fullPrice)
       (?fullPrice*(1-?discount) AS ?customerPrice)
WHERE { ?x ns:price ?p .
        ?x dc:title ?title .
        ?x ns:discount ?discount }
```

- **Result:**

?title	?fullPrice	?customerPrice
"The Semantic Web"	23	17.25
"SPARQL Tutorial"	42	33.6

Property Paths

- SPARQL 1.1 allows us to specify property paths in the place of a predicate in a triple pattern
- Property paths use regular expressions to enable us to write sophisticated queries that traverse an RDF graph
- Property paths allow for the more concise expression of some queries plus the ability to refer to paths of arbitrary length

Examples

- The / path operator denotes sequence
- **Query:** Find the name of any people that Alice knows

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

- **Query:** Find the name of people that Alice knows that are 2 "foaf:knows" links away

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

Example (cont'd)

- The same query can be written equivalently without property path expressions as follows:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x ?name
WHERE { ?x foaf:mbox <mailto:alice@example> .
        ?x foaf:knows ?a1 .
        ?a1 foaf:knows ?a2 .
        ?a2 foaf:name ?name . }
```


Examples

- The **+** operator denotes one or more occurrences of `foaf:knows`
- **Query:** Find all the people `:x` connects to via the `foaf:knows` relationship (using a path with an arbitrary length)

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

- The ***** operator denotes zero or more occurrences of `rdfs:subClassOf`
- **Query:** Find all types, including supertypes ,of each resource in the dataset

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
 SELECT ?x ?type
 WHERE { ?x rdf:type/rdfs:subClassOf* ?type }
```

SPARQL Endpoints

- SPARQL query processing service
- Supports the SPARQL protocol
- Issuing a SPARQL query is an HTTP GET request with parameter query
- Example:

```
GET /sparql?query=PREFIX+rd... HTTP/1.1
Host: dbpedia.org
User-agent: my-sparql-client/0.1
```

URL-encoded string with the SPARQL query

- Several Linked Data sets exposed via SPARQL endpoint: Send your query, receive the result!
  - DBpedia `http://dbpedia.org/sparql`
  - Musicbrainz `http://dbtune.org/musicbrainz/sparql`
  - World Factbook `http://www4.wiwiss.fu-berlin.de/factbook/snorql/`
  - LinkedMDB `http://data.linkedmdb.org/sparql`
  - `DBLP http://www4.wiwiss.fu-berlin.de/dblp/snorql/`

SPARQL Endpoints Query Result Formats

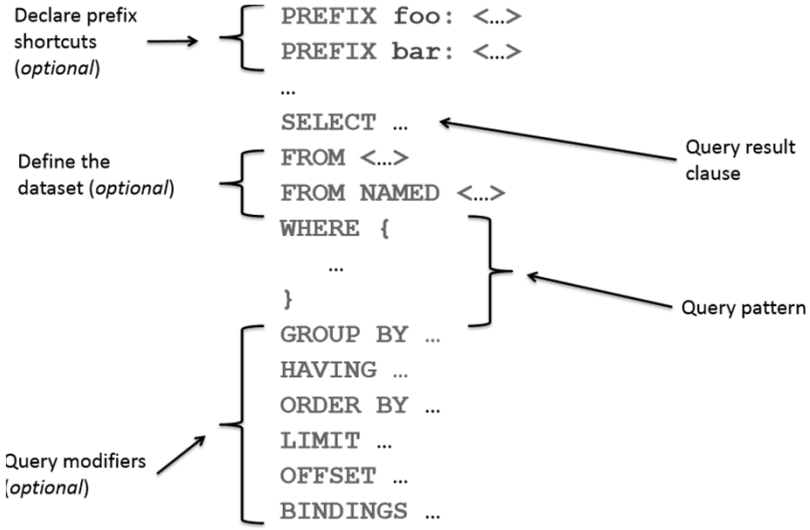
- For SELECT and ASK queries: XML, JSON, plain text
- For CONSTRUCT and DESCRIBE: RDF/XML, Turtle, ...
- How to request?
  - ACCEPT header

```
GET /sparql?query=PREFIX+rd... HTTP/1.1
Host: dbpedia.org
User-agent: my-sparql-client/0.1
Accept: application/sparql-results+json
```

  - Non-standard alternative: parameter out

```
GET /sparql?out=json&query=... HTTP/1.1
Host: dbpedia.org
User-agent: my-sparql-client/0.1
```

Summary SPARQL Query Syntax



Summary of SPARQL Filters

Category	Functions / Operators	Examples
Logical	`!, &&, ||, =, !=, <, <=, >, >=`	`?hasPermit || ?age < 25`
Math	`+, -, *, /`	`?decimal * 10 > ?minPercent`
Existence *(SPARQL 1.1)*	`EXISTS, NOT EXISTS`	`NOT EXISTS { ?p foaf:mbox ?email }`
SPARQL tests	`isURI, isBlank, isLiteral, bound`	`isURI(?person) || !bound(?person)`
Accessors	`str, lang, datatype`	`lang(?title) = "en"`
Miscellaneous	`sameTerm, langMatches, regex`	`regex(?ssn, "\\d{3}-\\d{2}-\\d{4}")`

Readings

- The SPARQL syntax & semantics can be found in the W3C specification www.w3.org/TR/sparql11-query (1.1) www.w3.org/TR/rdf-sparql-query (1.0)
- For formal semantics and expressive power of SPARQL read:
  - J. Perez, M. Arenas and C. Gutierrez. Semantics and Complexity of SPARQL. ACM Transactions on Database Systems, 34(3), Article 16 (45 pages), 2009
  - M. Arenas and J. Perez. Querying Semantic Web Data with SPARQL: State of the Art and Research Perspectives. In Proc. of PODS, Athens, Greece, pages 305-316, 2011
  - A. Mallea, M. Arenas, A. Hogan and A. Polleres. On Blank Nodes. Proc. of ISWC 2011
- Check out the SPARQL 1.1 Query Results Formats
  - JSON Format http://www.w3.org/TR/sparql11-results-json/
  - CSV and TSV Formats http://www.w3.org/TR/sparql11-results-csv-tsv/
  - XML Format http://www.w3.org/TR/rdf-sparql-XMLres/
- See www.w3.org/TR/2012/WD-sparql11-query-20120724/#propertypaths for the exact property path operators and syntax

Acknowledgements

- Manolis Koubarakis “An Introduction to SPARQL: Part I and II” lecture in course “Knowledge Technologies” http://cgi.di.uoa.gr/~pms509/
- Dieter Fensel, Federico Facca and Ioan Toma “Storage and Querying” lecture in course “Semantic Web” http://teaching-wiki.sti2.at/index.php/Semantic_Web
- Andreas Harth, Aidan Hogan, Spyros Kotoulas, Jacopo Urbani “Session 2a: SPARQL/Scalable RDF Indexing” Tutorial Series on Scalable Integration and Processing of Linked Data http://sild.cs.vu.nl/