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

Query Form:
- CONSTRUCT
- DESCRIBE
- SELECT
- ASK

Dataset Clause:
- FROM
- FROM NAMED

Where Clause (Graph Pattern):
- FILTER
- OPTIONAL
- AND
- UNION
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
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>`
  - ?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 BGP 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 BGP s, 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:

<table>
<thead>
<tr>
<th>?name</th>
<th>?mbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter Goodguy&quot;</td>
<td><a href="mailto:peter@example.org">mailto:peter@example.org</a></td>
</tr>
</tbody>
</table>
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 $P_1$ and then applying the keyword `OPTIONAL` to another graph pattern $P_2$ 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-syntax-ns#>.
  _:a rdf:type foaf:Person .
  _:a foaf:name "Alice" .
  _:b rdf:type foaf:Person .
  _:b foaf:name "Bob" .```
- Query:
  OPTIONAL { ?x foaf:mbox ?mbox } }
```
- Result:
  ```| ?name     | ?mbox                |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td><a href="mailto:alice@example.com">mailto:alice@example.com</a></td>
</tr>
<tr>
<td>&quot;Alice&quot;</td>
<td><a href="mailto:alice@work.example">mailto:alice@work.example</a></td>
</tr>
</tbody>
</table>
  | "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:**

```sparql
@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:creator "Bob" .
_:c dc10:title "SPARQL" .
_:c dc11:title "SPARQL (updated)" .
```

**Query:**

```sparql
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
{?
}
```

**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
UNION
```

**Result:**

<table>
<thead>
<tr>
<th>?author</th>
<th>?title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Bob&quot;</td>
<td>&quot;SPARQL Protocol Tutorial&quot;</td>
</tr>
<tr>
<td>&quot;Alice&quot;</td>
<td>&quot;SPARQL Query Language Tutorial&quot;</td>
</tr>
</tbody>
</table>

Queries with 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 .
```

**Queries:**

- SELECT ?v WHERE { ?v ?p "cat" }
- SELECT ?v WHERE { ?v ?p "cat"@en }

The queries 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:
  - \( P_1 \cdot P_2 \cdot \text{FILTER} (\ldots \text{boolean expression}\ldots) \)
- 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:
  ```ruby
  @prefix dc: <http://purl.org/dc/elements/1.1/> .
  @prefix : <http://example.org/book/> .
  @prefix ns: <http://example.org/ns#> .
  :book1 ns:price 42 .
  
  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) }
  
  Result:
  
<table>
<thead>
<tr>
<th>?title</th>
<th>?price</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The Semantic web&quot;</td>
<td>23</td>
</tr>
</tbody>
</table>
  ```
Blank Nodes in Query Results

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

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

Result:
```
<table>
<thead>
<tr>
<th>?x</th>
<th>?name</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:c</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>_:d</td>
<td>&quot;Bob&quot;</td>
</tr>
</tbody>
</table>
```

Blank Nodes in Query Results (cont’d)

Data:
```turtle
@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:
```sql
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:
```
<table>
<thead>
<tr>
<th>?x</th>
<th>?name</th>
<th>?y</th>
<th>?name</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:c</td>
<td>&quot;Alice&quot;</td>
<td>_:d</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>_:d</td>
<td>&quot;Bob&quot;</td>
<td>_:c</td>
<td>&quot;Alice&quot;</td>
</tr>
</tbody>
</table>
```
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:**
  ```
<table>
<thead>
<tr>
<th>?name1</th>
<th>?name2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>&quot;Alice&quot;</td>
</tr>
</tbody>
</table>
  ```

A Complete SPARQL Example

```sparql
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/>

SELECT ?name ?expertise
FROM <http://data.uoc.gr/>
WHERE {
  ?person foaf:title ?title . FILTER regex(?title, "^Prof")
  OPTIONAL {
    ?person oo:availableToCommentOn ?expertiseURI .
    ?expertiseURI rdfs:label ?expertise
  }
}
ORDER BY ?surname
```

*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/>

WHERE {
  ?person foaf:title ?title . FILTER regex(?title, "^ Prof")
  OPTIONAL {
    ?person oo:availableToCommentOn ?expertiseURI .
    ?expertiseURI rdfs:label ?expertise
  }
}
```

```
ORDER BY ?surname

25
```

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
FROM <http://data.uoc.gr/>
WHERE {
  ?person foaf:title ?title . FILTER regex(?title, "^ Prof")
  OPTIONAL {
    ?person oo:availableToCommentOn ?expertiseURI .
    ?expertiseURI rdfs:label ?expertise
  }
}
```

```
ORDER BY ?surname

26
```
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/>

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/>

SELECT ?name ?expertise
FROM <http://data.uoc.gr/>
WHERE {
?person foaf:title ?title . FILTER regex(?title, "^Prof")
OPTIONAL {
?person oo:availableToCommentOn ?expertiseURI .
?expertiseURI rdfs:label ?expertise
}
}

ORDER BY ?surname

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 output results by surname (as you probably guessed)

...also...

**LIMIT**

Only return 10 results

```
ORDER BY ?surname LIMIT 10
```

**OFFSET**

Return results 20–30

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

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

- A mapping satisfies
  - $\mu_X = c$ if it gives the value $c$ to variable $?X$
  - $?X = ?Y$ if it gives the same value to $?X$ and $?Y$
  - $\text{bound}(?X)$ if it is defined for $?X$
SPARQL Algebraic Operators

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

$$\Omega = \begin{array}{cc}
?x & ?y \\
 a & b \\
a & c \\
d & e \\
\end{array}$$

$$\pi_{?x}(\Omega) = \begin{array}{c}
?x \\
a \\
\end{array}$$

- $\mu_1$ \quad $\text{card}(\mu_1) = 2$
- $\mu_2$ \quad $\text{card}(\mu_2) = 1$
SPARQL Algebraic Operators

- SPARQL algebra defines 5 operators on mapping bags
  - Unary ops:
    - $\pi$ (projection),
    - $\sigma$ (selection, also called filtering)
  - Binary ops:
    - $\cup$ (union)
    - $\bowtie$ (join)
    - $\bowtie\bowtie$ (optional)

$\Omega$

<table>
<thead>
<tr>
<th>?x</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>?x</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>

$\mu_1$, $\text{card}(\mu_1) = 1$

$\mu_2$, $\text{card}(\mu_2) = 1$

$\Omega_1 \cup \Omega_2$

$\Omega_1$

<table>
<thead>
<tr>
<th>?x</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

$\mu_1$

$\Omega_2$

<table>
<thead>
<tr>
<th>?x</th>
<th>?z</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

$\mu_2$

$\Omega_1 \cup \Omega_2$

<table>
<thead>
<tr>
<th>?x</th>
<th>?y</th>
<th>?z</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>d</td>
</tr>
</tbody>
</table>

$\mu_1$

$\mu_2$

$\Omega$ is unbound in $\mu_1$
SPARQL Algebraic Operators

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

\[ \Omega_1 \bowtie\triangleright \Omega_2 \]

<table>
<thead>
<tr>
<th>$\Omega_1$</th>
<th>$\Omega_2$</th>
<th>$\Omega_1 \bowtie\triangleright \Omega_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>?x</td>
<td>?y</td>
<td>$\mu_1$</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>$\mu_2$</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>$\mu_3$</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>$\mu_4$</td>
</tr>
<tr>
<td>?y</td>
<td>?z</td>
<td>$\mu_5 = \mu_1 \cup \mu_4$</td>
</tr>
<tr>
<td>b</td>
<td>f</td>
<td>$\mu_6 = \mu_3 \cup \mu_4$</td>
</tr>
</tbody>
</table>

$\Omega_1 \bowtie\triangleright \Omega_2$ can be written as:

- $\Omega_1 \bowtie \Omega_2$ (a, b, f): (P₁ OPTIONAL P₂)
- (P₁ AND P₂) UNION (P₁ MINUS P₂)

\[ \Omega_1 \bowtie\triangleright \Omega_2 \]

<table>
<thead>
<tr>
<th>$\Omega_1$</th>
<th>$\Omega_2$</th>
<th>$\Omega_1 \bowtie\triangleright \Omega_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>?x</td>
<td>?y</td>
<td>$\mu_1$</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>$\mu_2$</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>$\mu_3$</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>$\mu_4$</td>
</tr>
<tr>
<td>?y</td>
<td>?z</td>
<td>$\mu_5 = \mu_1 \cup \mu_3$</td>
</tr>
<tr>
<td>b</td>
<td>f</td>
<td>$\mu_6 = \mu_3 \cup \mu_2$</td>
</tr>
</tbody>
</table>

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

(c, d, -): $\Omega_1 \setminus \Omega_2$
Few SPARQL Algebraic Equivalences

\[ \sigma_{x = ?x}(P_1) \equiv P_1 \quad P_1 \otimes \emptyset \equiv \emptyset \quad P_1 \cup \emptyset \equiv P_1 \\
P_1 \cup P_2 \equiv P_2 \cup P_1 \quad P_1 \otimes P_2 \equiv P_2 \otimes P_1 \\
P_1 \cup (P_2 \cup P_3) \equiv (P_1 \cup P_2) \cup P_3 \\
P_1 \otimes (P_2 \otimes P_3) \equiv (P_1 \otimes P_2) \otimes P_3 \\
P_1 \otimes (P_2 \cup P_3) \equiv (P_1 \otimes P_2) \cup (P_1 \otimes P_3) \\
\]

\[ P_1 \setminus P_1 \equiv \emptyset \quad P_1 \setminus (P_2 \cup P_3) \equiv (P_1 \setminus P_2) \setminus P_3 \\
P_1 \otimes (P_2 \setminus P_3) \equiv (P_1 \otimes P_2) \setminus P_3 \\
(P_1 \setminus (P_1 \setminus P_2)) \cup (P_1 \setminus P_2) \equiv P_1 \]

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
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:**
  ```reasoning
  @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
  ```reasoning
  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:**
  ```text
  ?name
  "Bob"
  "Mike"
  ```

Examples with Negation

- **Query:** Find people with a name but **no** expressed age:
  ```reasoning
  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:**
  ```text
  ? 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
  
  ```sparql
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  PREFIX ex: <http://example.org/schema/>
  
  SELECT ?name
  WHERE { ?x foaf:givenName ?name .
                       FILTER(?age >= 60) } .
           FILTER (!bound(?age))
  }
  ```

- **Result:**
  
<table>
<thead>
<tr>
<th>?name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
</tr>
</tbody>
</table>

Examples with Negation (cont’d)

- **Note:** 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
  
  ```sparql
  { 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 $\varphi$ and $\theta$ two arbitrary FOL formulas.
  - If $KB$ entails $\varphi$ then $KB$ union $\{ \theta \}$ entails $\varphi$ 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.

OWA vs. CWA in RDF

- 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.
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/>.

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)
  ?book :price ?lprice. }
GROUP BY ?org
HAVING (SUM(?lprice) > 10)

Result:
<table>
<thead>
<tr>
<th>?org</th>
<th>?totalPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://books.example/org1">http://books.example/org1</a></td>
<td>21</td>
</tr>
</tbody>
</table>
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 $P_3$:
  
  $P_1$.
  
  $\{ \text{SELECT} \}$
  
  $\text{WHERE \{P_2}\}$
  
  $\}$
  
  $P_3$.

- 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:
  
  ```sparql
@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

```sparql
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:**

<table>
<thead>
<tr>
<th>?c</th>
<th>?increase</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://sales.com/c2">http://sales.com/c2</a></td>
<td>&quot;4000&quot;^^<a href="http://www.w3.org/2001/XMLSchema#integer">http://www.w3.org/2001/XMLSchema#integer</a></td>
</tr>
</tbody>
</table>

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
    - $P_1 \text{ MINUS } P_2$ 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
    - $P_1 \text{ NOT EXISTS } P_2$ returns true if the pattern $P_2$ do not matches the data after substituting all mappings in $P_1$ (shared vars)
- Proposition: $(P_1 \text{ MINUS } P_2)$ is equivalent to: $P_1 \text{ OPT } (P_2 \text{ AND } (?X_1, ?X_2, ?X_3))$ FILTER $\neg\text{bound}(?X_1)$, where $?X_1, ?X_2, ?X_3$ are mentioned neither in $P_1$ nor in $P_2$
- 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 
    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:
  ```sparql
  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 $P_2$
  ```sparql
  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

<table>
<thead>
<tr>
<th>s</th>
<th>p</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://example/a">http://example/a</a></td>
<td><a href="http://example/b">http://example/b</a></td>
<td><a href="http://example/c">http://example/c</a></td>
</tr>
</tbody>
</table>

See more examples
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 ns:price 42 .
  :book1 ns:discount 0.2 .
  :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) 
           ?x ns:discount ?discount} 

- **Result:**

<table>
<thead>
<tr>
<th>?title</th>
<th>?price</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The Semantic Web&quot;</td>
<td>17.25</td>
</tr>
<tr>
<td>&quot;SPARQL Tutorial&quot;</td>
<td>33.6</td>
</tr>
</tbody>
</table>

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) 
           ?x ns:discount ?discount } 

- **Result:**

<table>
<thead>
<tr>
<th>?title</th>
<th>?fullPrice</th>
<th>?customerPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The Semantic Web&quot;</td>
<td>23</td>
<td>17.25</td>
</tr>
<tr>
<td>&quot;SPARQL Tutorial&quot;</td>
<td>42</td>
<td>33.6</td>
</tr>
</tbody>
</table>
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
  
  ```sparql
  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
  
  ```sparql
  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:

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

  ```sparql
  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

  ```sparql
  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 URL-encoded string with the SPARQL query
- Example:
  
  ```
  GET /sparql?query=PREFIX+rd... HTTP/1.1
  Host: dbpedia.org
  User-agent: my-sparql-client/0.1
  ```

- 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

 Declare prefix shortcuts (optional)

{PREFIX foo: <...>
 PREFIX bar: <...>

...}

SELECT ...

FROM <...>

WHERE 

...

GROUP BY ...

HAVING ...

ORDER BY ...

LIMIT ...

OFFSET ...

BINDINGS ...

### Summary of SPARQL Filters

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions / Operators</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>¬, `,</td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td>+, -, *, /</td>
<td>?decimal * 10 &gt; ?minPercent</td>
</tr>
<tr>
<td>Existence</td>
<td>EXISTS, NOT EXISTS</td>
<td>NOT EXISTS ( ?p foaf:mbox ?email )</td>
</tr>
<tr>
<td>SPARQL tests</td>
<td>isURI, isBlank, isLiteral, bound</td>
<td>isURI(?person)</td>
</tr>
<tr>
<td>Accessors</td>
<td>str, lang, datatype</td>
<td>lang(?title) = &quot;en&quot;</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>sameTerm, langMatches, regex</td>
<td>regex(?ssn, &quot;\d{3}-\d{2}-\d{4}&quot; )</td>
</tr>
</tbody>
</table>
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:
- 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/#proprietary for the exact property path operators and syntax

Acknowledgements

- 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/