SPARQL Protocol and RDF Query Language (SPARQL)
Semantic Web (CSC688 P)

Ubbo Visser (Instructor)    Saminda Abeyruwan (Research Assistant)

Department of Computer Science
University of Miami

November 8, 2011
1. Announcements

2. In retrospect

3. Query types

4. Basics
Reading

- 7.1.1-7.1.8 [HKR09]

Acknowledgement

- Most of the examples in this lecture slides are borrowed from SPARQL Query Language for RDF

Fall 2011

- Project presentations will be conducted on Nov 29\textsuperscript{th} (class time), Dec 1\textsuperscript{st} (class time), and Dec 2\textsuperscript{nd} (4 pm - 6 pm).
- It is 20 minutes long (timing will be strict) plus a few questions.
- You present 3 presentations on Nov 29\textsuperscript{th}, 4 presentations Dec 1\textsuperscript{st}, and 4 presentations on Dec 2\textsuperscript{nd}.
- The order of the presentations will be announced soon.
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```

```
Orphan ⊑ Human ⊓ ∀ hasParent. ¬ Alive
```
Query types

1. Retrieve instances.

   ```sql
   SELECT ?x WHERE {
     ?x rdf:type family:Person .
   }
   ```

2. Retrieve subclasses.

   ```sql
   SELECT ?x WHERE {
     ?x rdfs:subClassOf family:Person .
   }
   ```

3. Retrieve subclasses, and their instances.

   ```sql
   SELECT ?x ?y WHERE {
     ?y rdfs:subClassOf family:Person .
   }
   ```
I have written a simple framework to query the knowledge base using Jena and Pellet API. It is available in the class web site.

```java
public interface OwlHelper {
    InfModel loadInfModel(File kb);
    InfModel loadInfModel(File tBox, File aBox);
    void startReasoner(InfModel model);
    void execQuery(String query, Model model, ResultSetCallback callback);
}

public interface ResultSetCallback {
    void run(ResultSet resultSet, Query query);
}
```

You can use this framework in your code as follows:

```java
OwlHelper helper = OwlHelperFactory.createDefaultOwlHelper();
InfModel model = helper.loadInfModel(kbFile);
helper.startReasoner(model);
...
String query = ...;
helper.execQuery(query, model, new ResultSetFormatterCallback());
```
**SPARQL**

- W3C recommendation for querying RDF and RDFS.
- We can use SPARQL to a certain extent to query OWL 2 DL knowledge bases. But the preferred way is *conjunctive queries*.

**Graph patterns**

- SPARQL is based on matching *graph patterns* w.r.t RDF, RDFS (supported features), or OWL (supported features) graphs.
- A *graph pattern* is similar to *triple pattern*, but with the option of *variables* in subject, predicate or object. e.g.,

  ```
  ```

- ?parent is a **variable**. This variable could also be written as $parent.
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.

- Therefore, BGP is a conjunction of triple patterns. e.g.,

\[
\begin{align*}
\text{?x} &<\text{http://family.org/family.owl#hasParent}>\text{ ?parent} . \\
\text{?x} &<\text{http://family.org/family.owl#hasUncle}>\text{ ?uncle}
\end{align*}
\]

- There is no keyword for conjunction in SPARQL.
A group graph pattern is a set of graph patterns delimited with braces. e.g.,

```
{  
    { ?x <http://family.org/family.owl#hasParent> ?y . }  
    { ?x <http://family.org/family.owl#hasUncle> ?z . }  
    { }  
}
```

- `{ }` is the empty group graph pattern.
- Group graph patterns are used with other constructors, which we will see in few slides.
Major query parts

- **PREFIX**: declares the namespace prefix,
- **SELECT**: determines the general result format, and
- **WHERE**: actual query is initiated with group graph patterns.

The result of a query is a set of *bindings* for the variables appearing in the **SELECT** clause. These bindings are shown in tabular format.

**SELECT** and **WHERE** clauses are like in SQL. But keep in mind that SPARQL and SQL are very different languages.
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX family: <http://family.org/family.owl#>

SELECT ?x
WHERE
{
  family:daughter family:hasParent ?x .
}

-----------------
| x |
-----------------
| family:mother |
| family:father |
-----------------
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX family: <http://family.org/family.owl#>

SELECT ?x ?y ?z
{
  ?x family:hasParent ?y .
  ?x family:hasUncle ?z
}

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>family:daughter</td>
<td>family:mother</td>
<td>family:uncle</td>
</tr>
<tr>
<td>family:daughter</td>
<td>family:father</td>
<td>family:uncle</td>
</tr>
<tr>
<td>family:son</td>
<td>family:mother</td>
<td>family:uncle</td>
</tr>
<tr>
<td>family:son</td>
<td>family:father</td>
<td>family:uncle</td>
</tr>
</tbody>
</table>
Queries with literals

- We have careful when matching literals. E.g.,

  ```sql
  SELECT ?x WHERE { ?x ?p "Ubbo" .}
  ```

  and

  ```sql
  SELECT ?x WHERE { ?x ?p "Ubbo"@en .}
  ```

  have different results.

- xsd data types:

  ```sql
  SELECT ?x WHERE { ?x ?hasAge "38"^^xsd:nonNegativeInteger .}
  ---------------------
  | x                  |
  ---------------------
  | family:father      |
  ---------------------
  or
  SELECT ?x WHERE { ?x ?hasAge 38.}
  ```
Queries with literals - continued

"chat"

'chat'@fr with language tag "fr"

"xyz"^^<http://example.org/ns/userDatatype>

"abc"^^appNS:appDataType

"'The librarian said, "Perhaps you would enjoy 'War and Peace'."'"

1, which is the same as "1"^^xsd:integer

1.3, which is the same as "1.3"^^xsd:decimal

1.300, which is the same as "1.300"^^xsd:decimal

1.0e6, which is the same as "1.0e6"^^xsd:double

true, which is the same as "true"^^xsd:boolean

false, which is the same as "false"^^xsd:boolean
Blank nodes in query results

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

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

<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 graph patterns

- Blank nodes assert the existence of a corresponding element in the input graph, but they do not provide any information about the identity of this element.
- 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 term.
**Constrains on variables**

- **FILTER** restricts variable bindings to those for which the filter expression evaluates to *true*.

```sparql
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix : <http://example.org/book/> .
@prefix ns: <http://example.org/ns#> .


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)  
}

=> "The Semantic Web"  23
```
Constrains on variables

- **Regular expression filter:**

```prefix
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title
WHERE
{
  ?x dc:title ?title
  FILTER regex(?title, "^SPARQL")
}
```

=> SPARQL Tutorial

**SPARQL Tutorial**

- **Group graph patterns** are used to restrict the scope of the `FILTER`.
- `FILTER` is a restriction on solutions over the whole group in which it appears.
- One can have multiple `FILTER` conditions in a group graph pattern. The result is equivalent to a single filter with conjuncted filter conditions.
- `FILTER` can have very complex boolean conditions.
These graph patterns have same set of solutions

```
{  
    ?x foaf:name ?name .  
    FILTER regex(?name, "Smith")  
}

{  
    FILTER regex(?name, "Smith")  
    ?x foaf:name ?name .  
}

{  
    ?x foaf:name ?name .  
    FILTER regex(?name, "Smith")  
}
```
OPTIONAL graph patterns

- With OWA, the complete structures cannot be assumed in all RDF graphs. This of the ABox.
- Therefore, we need a way to extract the available information, even though some part of the query pattern does not match.
- OPTIONAL provides this facility. If the graph pattern does not match, it does not create bindings, but does not eliminate the solution as well.

KB

```ruby
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

_:a rdf:type foaf:Person .
_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@example.com> .
_:a foaf:mbox <mailto:alice@work.example> .
_:b rdf:type foaf:Person .
_:b foaf:name "Bob" .
```
**OPTIONAL example**

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

<table>
<thead>
<tr>
<th>name</th>
<th>mbox</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>
<tr>
<td>&quot;Bob&quot;</td>
<td></td>
</tr>
</tbody>
</table>
**OPTIONAL properties**

- Normally, we start with a graph pattern $P_1$ and then apply **OPTIONAL** to another graph pattern $P_2$ that follows it.

  ```plaintext
  P1 OPTIONAL { P2 }
  ```

- **OPTIONAL** is a binary operator.

- **OPTIONAL** is left-associative.

  ```plaintext
  P1 OPTIONAL { P2 } OPTIONAL { P3 }
  <=>
  { P1 OPTIONAL { P2 } } OPTIONAL { P3 }
  ```

  ```plaintext
  { OPTIONAL { P } }
  <=>
  { { } OPTIONAL { P } }
  ```

- **OPTIONAL** has higher precedence than conjunction.
**FILTER in OPTIONAL**

- The group graph pattern following the OPTIONAL can be as complex as possible.

---

**KB**

```sparql
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix : <http://example.org/book/> .
@prefix ns: <http://example.org/ns#> .

:book2 ns:price 42 .
```
**FILTER in OPTIONAL example**

```sparql
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
SELECT ?title ?price
WHERE {
    OPTIONAL {
        ?x ns:price ?price .
        FILTER (?price < 30)
    }
}
```

<table>
<thead>
<tr>
<th>title</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SPARQL Tutorial&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;A New SPARQL Tutorial&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;The Semantic Web&quot;</td>
<td>23</td>
</tr>
</tbody>
</table>
### Multiple OPTIONAL

```sparql
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:a foaf:homepage <http://work.example.org/alice/> .
_:b foaf:name "Bob" .
_:b foaf:mbox <mailto:bob@work.example> .

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

<table>
<thead>
<tr>
<th>name</th>
<th>mbox</th>
<th>hpage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td></td>
<td><a href="http://work.example.org/alice/">http://work.example.org/alice/</a></td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td><a href="mailto:bob@work.example">mailto:bob@work.example</a></td>
<td></td>
</tr>
</tbody>
</table>
Example

@prefix ex: <http://example.org/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ns: <http://example.org/ns#> .


ex:book3 ns:price 34 .

PREFIX ex: <http://example.org/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>

**********************************************
SELECT ?book ?title
  { ?book ns:price ?price .}
}

**********************************************
SELECT ?book ?title
    OPTIONAL { ?book dc:title ?title .} }
  { ?book ns:price ?price .}
}

**********************************************
SELECT ?book ?title
    OPTIONAL { ?book dc:title ?title .} }
  { ?book ns:price ?price .}
}

**********************************************
UNION

- **UNION** provides the facility to form *disjunction of graph patterns*, such that one of several graph patterns may match. All the alternative matching patterns are returned.

KB

```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)" .
```
UNION example

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

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SPARQL Protocol Tutorial&quot;</td>
</tr>
<tr>
<td>&quot;SPARQL&quot;</td>
</tr>
<tr>
<td>&quot;SPARQL (updated)&quot;</td>
</tr>
<tr>
<td>&quot;SPARQL Query Language Tutorial&quot;</td>
</tr>
</tbody>
</table>
UNION example

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

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

- **UNION** is a binary operator.
- Group graph patterns are evaluated independently and combine the results using *set theoretic union*.
- We have to decide whether to use the same variable in each alternative, as this decision provides different results.

**UNION example**

```
SELECT ?x ?y
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;SPARQL (Updated)&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;SPARQL Protocol ...&quot;</td>
</tr>
<tr>
<td>&quot;SPARQL&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;SPARQL Query ...&quot;</td>
<td></td>
</tr>
</tbody>
</table>
```
Properties of \texttt{UNION}

- \texttt{UNION} is left-associative.
- \texttt{UNION} and \texttt{OPTIONAL} have same precedence.
- \texttt{UNION} has higher precedence than conjunction.
- **Commutative**

\[
P \text{ UNION } Q \iff Q \text{ UNION } P
\]

- **Associative**

\[
\{P \text{ UNION } Q\} \text{ UNION } R \iff P \text{ UNION } \{Q \text{ UNION } R\}
\]
OPTIONAL, UNION examples

```
{ {s1 p1 o1} UNION {s2 p2 o2} 
  OPTIONAL {s3 p3 o3} }
<=>
{ { {s1 p1 o1} UNION {s2 p2 o2} } 
  OPTIONAL {s3 p3 o3} }
```

```
{ {s1 p1 o1} OPTIONAL {s2 p2 o1} 
  UNION {s3 p3 o3} OPTIONAL 
  {s4 p4 o4} OPTIONAL 
  {s5 p5 o5} }
<=>
{ { { {s1 p1 o1} OPTIONAL {s2 p2 o1} 
       } UNION {s3 p3 o3} 
       } OPTIONAL {s4 p4 o4} 
    } OPTIONAL {s5 p5 o5} 
  }
```
UNION and conjunction

```plaintext
{ {s1 p1 o1} UNION {s2 p2 o1}
  {s3 p3 o3}
}

<=>
{ { {s1 p1 o1} UNION {s2 p2 o1}
  }{s3 p3 o3}
}
```
@prefix ex: <http://example.org/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ns: <http://example.org/ns#> .


Example

```sparql
PREFIX ex: <http://example.org/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
WHERE {
  UNION 
  { ?book dc:creator ex:jones . }
  { ?book ns:price ?price . }
}
```

<table>
<thead>
<tr>
<th>book</th>
<th>title</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://example.org/book3">http://example.org/book3</a></td>
<td></td>
<td>35</td>
</tr>
<tr>
<td><a href="http://example.org/book2">http://example.org/book2</a></td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>
Example

PREFIX ex: <http://example.org/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
WHERE
{
  UNION
}

<table>
<thead>
<tr>
<th>book</th>
<th>title</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://example.org/book1">http://example.org/book1</a></td>
<td>&quot;Semantic Web&quot;</td>
<td></td>
</tr>
<tr>
<td><a href="http://example.org/book3">http://example.org/book3</a></td>
<td></td>
<td>35</td>
</tr>
<tr>
<td><a href="http://example.org/book2">http://example.org/book2</a></td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>
More about `FILTER` and special operators

- **FILTER supports** `=, >, <, ≥, ≤, and !=` operators.
- Each operator is defined for all datatype that SPARQL supports. e.g., `xsd:dateTime`
- All literals that have different datatypes are not compatible with prior operators, but `=` and `!=`.
- But, they produce an error if unknown datatypes are given.
- Multiple filter conditions are combined with `&&` (logical `and`), `||` (logical `or`) and `!` (logical `not`).
- Conjunction: can be expressed with multiple `FILTER` within one graph pattern.
- Disjunction: a graph pattern could be split into multiple alternative patterns that use equal conditions with one of filter part.
- Supports numerical operators, `+`, `-`, `*`, and `/`, only if the variable are bounded in a meaningful way.
## Unary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BOUND(A)</code></td>
<td>true if A is a bounded variable</td>
</tr>
<tr>
<td><code>isURI(A)</code></td>
<td>true if A is a URI</td>
</tr>
<tr>
<td><code>isBLANK(A)</code></td>
<td>true if A is a blank node</td>
</tr>
<tr>
<td><code>isLITERAL(A)</code></td>
<td>true if A is a RDF literal</td>
</tr>
<tr>
<td><code>STR(A)</code></td>
<td>maps RDF literals or URIs to the corresponding lexical representation of type <code>xsd:string</code></td>
</tr>
<tr>
<td><code>LANG(A)</code></td>
<td>returns language code of an RDF literal as <code>xsd:string</code>, or an empty string if no such setting is specified</td>
</tr>
<tr>
<td><code>DATATYPE(A)</code></td>
<td>returns the URI of an RDF literal datatype of the value “<code>xsd:string</code>” for untyped literals without language setting; not applicable to literals with language setting</td>
</tr>
<tr>
<td><code>sameTERM(A,B)</code></td>
<td>true if A and B are the same RDF terms (direct term comparison)</td>
</tr>
<tr>
<td><code>langMATCHES(A,B)</code></td>
<td>true if the literal A is a language tag that matches the pattern B</td>
</tr>
<tr>
<td><code>REGEX(A,B)</code></td>
<td>true if the regular expression B can be matched to the string A</td>
</tr>
</tbody>
</table>
Example

PREFIX ex: <http://example.org/>
SELECT ?book
WHERE
{
  { ?book ex:isPublishedBy <http://crc-press.com/uri> . }
  OPTIONAL { ?book ex:author ?author .}
  FILTER( DATATYPE(?author) = <http://www.w3.org/2001/XMLSchema#string> )
}

PREFIX ex: <http://example.org/>
SELECT ?book
WHERE
{
  FILTER( REGEX(?title, "^Foundations of") )
}
Query forms

- Tabular representation is useful for processing results sequentially.
- If the structure and mutual relations of objects in the results set are more important, RDF representation of the results is more appropriate.
- **CONSTRUCT** returns RDF graph specified by a graph template.
- **ASK** tests whether or not a graph pattern has a solution. This returns whether or not a solution exists.
Example **CONSTRUCT**

```turtle
@prefix ex: <http://example.org/> .

ex:alice ex:email "alice@example.org" .
ex:alice ex:email "a.miller@example.org" .
ex:alice ex:phone "123456789" .
ex:alice ex:phone "987654321" .

PREFIX ex: <http://example.org/>
CONSTRUCT {
  _id1 ex:email ?email .
  _id1 ex:phone ?phone .
  _id1 ex:person ?person .
}
WHERE
{
}

_y ex:email "alice@example.org";
...
```
Example ASK

PREFIX ex: <http://example.org/>

ASK
{
}

=> TRUE
Modifiers

- **To narrow down the result set.**
- **Modifiers controls the details regarding the form and size of result lists.**
- **Most constructs affects only results obtained with SELECT.**

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ORDER BY</strong></td>
<td>sort in ascending order based on the meaningful bounded variable.</td>
</tr>
<tr>
<td><strong>DESC</strong></td>
<td>sort by descending order</td>
</tr>
<tr>
<td><strong>ASC</strong></td>
<td>sort by ascending order</td>
</tr>
<tr>
<td><strong>LIMIT</strong></td>
<td>maximum results</td>
</tr>
<tr>
<td><strong>OFFSET</strong></td>
<td>staring position for piecewise retrieval of results</td>
</tr>
<tr>
<td><strong>DISTINCT</strong></td>
<td>remove repetitions from result set</td>
</tr>
</tbody>
</table>
Order of application

- All the parameters are allowed to be combined. Therefore, SPARQL defines the following processing steps:
  - Sort results based on ORDER BY.
  - Remove non selected variables from the result set (projection).
  - Remove duplicate results.
  - Remove the number of initial results as specified by OFFSET.
  - Remove all results after the number specified by LIMIT.

- **LARQ**: combination of ARQ and Lucene. This is a specific example.
Examples

PREFIX ex: <http://example.org/>
SELECT ?book ?price
WHERE
{
}
ORDER BY ?price

SELECT ?book ?price
WHERE
{
}
ORDER BY ASC(?price)

SELECT *
WHERE
{
}
ORDER BY ?s LIMIT 5 OFFSET 25
The Manchester OWL

**DL Query**
- Searching in a classified ontology using Manchester OWL syntax.
- It is based on OWL abstract syntax and DL style syntax.
- **Supports** some, only, value, min, exactly, max, and, or, and not.
- Supports data values and datatypes with XSD facets.
- Lets see an example based on photography ontology (OWL 2).
Example

- Which equipment can reduce blur?
  Equipment and reduce some Blur

- What types of lens is a 35-120mm?
  Lens and (hasMinEffectiveFocalLength value 35)
  and (hasMaxEffectiveFocalLength value 120)

- Which adjustments can I use to increase the exposure without affecting the depth of field?
  Adjustment and increase some ExposureLevel and not(affects some DepthOfField)
Example

- Person and hasAge some nonNegativeInteger
- Person and hasAge some int[>40]
- Person and hasAge some int[>10,<40]