

INFO216: **Advanced Modelling**

Theme, spring 2018:
**Modelling and Programming
the Web of Data**

Andreas L. Opdahl
<Andreas.Opdahl@uib.no>



Session 3: SPARQL

- Themes:
 - **introducing SPARQL Update**
 - SPARQL 1.1 Update Language (older: “SPARUL”)
 - programming SPARQL updates
 - **repetition of SPARQL queries**
 - ...plus a few new things:
 - composite properties, other query types...
 - SPARQL 1.1
 - programming SPARQL queries



Session 3: SPARQL

- Themes:
 - **introducing SPARQL Update**
 - SPARQL 1.1 Update Language (older: “SPARUL”)
 - programming SPARQL updates
 - **repetition of SPARQL queries**
 - ...plus a few new things:
 - composite properties, other query types...
 - SPARQL 1.1
 - programming SPARQL queries



Readings

- Sources:
 - Allemang & Hendler (2011):
Semantic Web for the Working Ontologist,
chapter 5 (SPARQL)
 - W3C resources:
 - <http://www.w3.org/TR/sparql11-overview/>
 - <http://www.w3.org/TR/sparql11-query/>
 - <http://www.w3.org/TR/sparql11-update/>
 - *as usual: wiki.uib.no/info216*



SPARQL



SPARQL

- Simple/SPARQL Protocol and RDF Query Language

- Example:

```
SELECT ?child ?sister WHERE {  
    ?child fam:hasParent ?parent .  
    ?parent fadm:hasSister ?sister .  
}
```

- **Main idea:**

- we give SPARQL an underspecified RDF graph:
a **pattern...**
- some of the nodes are **variables**:
 - other nodes are **IRIs**, or **literal** values as before
- SPARQL tries to **match** the pattern to an RDF graph
- ...and returns each match as a **result**



SPARQL organisation

- SPARQL 1.1 comprises:
 - *Query Language* – the core of SPARQL
 - result formats: XML, JSON, CSV, TSV
 - federated queries
(involving several endpoints)
 - *SPARQL Update Language* (“SPARUL”) – since 1.1
 - service descriptions and protocols
 - test cases



SPARQL execution

- Defined in two parts:
 - SPARQL protocol: RDF over HTTP
 - SPARQL results: XML or JSON (or CSV, TSV...)
- SPARQL engine:
 - working on a native RDF data set
 - working on a wrapped (virtualised) non-RDF data set
 - e.g., a relational database wrapped by D2R
- **RDF and SPARQL offer a *common abstraction level* for data interoperability**



SPARQL queries and updates

- Two groups of *manipulations*:
 - graph store *updates*:
 - LOAD, CLEAR, INSERT, DELETE
 - graph store *management*:
 - CREATE, DROP, COPY, MOVE, ADD
- Four types of dataset *queries*:
 - SELECT: returns table
 - ASK: returns yes/no
 - CONSTRUCT: returns a graph
 - DESCRIBE: returns a graph
- *All are written in TURTLE-like style*
 - *...the variations are similar too*



SPARQL Update



Management of graph stores

- CREATE GRAPH <<http://example.org/testGraph>>
CREATE SILENT GRAPH <<http://example.org/testGraph>>
 - creates a new empty graph with the specified name in the graph store
- DROP GRAPH <<http://example.org/testGraph>>
DROP DEFAULT
DROP NAMED
DROP ALL
 - removes the specified graph(s) from the graph store



Management of graph stores

- COPY <http://example.org/testGraph> TO DEFAULT
COPY DEFAULT TO <http://example.org/otherGraph>
COPY <http://example.org/otherGraph> TO
<http://example.org/yetAnotherGraph>
 - inserts all data from the source into the destination graph
 - source graph not affected
 - destination graph emptied before insertion
- MOVE <http://example.org/testGraph> TO DEFAULT
 - like COPY, but source graph is removed afterwards
- ADD <http://example.org/testGraph> TO DEFAULT
 - like COPY, but destination graph is not emptied beforehand



Updates: LOAD and CLEAR

- Read an RDF document from an IRL:
LOAD <file:/home/andreas/myGraph.ttl>
LOAD <file:/home/andreas/myGraph.ttl> INTO GRAPH
 <http://example/org/testGraph>
- Remove all triples from a graph in the graph store:
CLEAR GRAPH <http://example/org/testGraph>



Updates: INSERT DATA

- Create new triples, e.g.:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
INSERT DATA
```

```
{  
  <http://example/book3> dc:title "A new book" ;  
  dc:creator "A.N.Other" .  
}
```

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
PREFIX ns: <http://example.org/ns#>
```

```
INSERT DATA
```

```
{  
  GRAPH <http://example/bookStore>  
  { <http://example/book1> ns:price 42 }  
}
```

- *Suitable for creating small data sets...*



Updates: DELETE DATA

- Remove given triples, e.g.:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
DELETE DATA
{
    <http://example/book2> dc:title "David Copperfield" ;
    dc:creator "Edmund Wells" .
}
```

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
DELETE DATA
{
    GRAPH <http://example/bookStore> {
        <http://example/book2> dc:title "David Copperfield" ;
        dc:creator "Edmund Wells" .
    }
}
```



Updates: DELETE/INSERT from pattern

- Remove and create triples from patterns, e.g.:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
WITH <http://example/addresses>
```

```
DELETE { ?person foaf:givenName 'Bill' }
```

```
INSERT { ?person foaf:givenName 'William' }
```

```
WHERE {
```

```
    ?person foaf:givenName 'Bill'
```

```
}
```

- WITH specifies the named graph to modify



Updates: DELETE from pattern

- Create triples from pattern, e.g.:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

```
WITH <http://example/bookStore>
```

```
DELETE {
```

```
    ?book ?p ?v
```

```
} WHERE {
```

```
    ?book dc:date ?date .
```

```
    FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
```

```
    ?book ?p ?v
```

```
}
```



Updates: INSERT from pattern

- Create triples from pattern, e.g.:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

```
INSERT {  
  GRAPH <http://example/bookStore2> {  
    ?book ?p ?v  
  }  
} WHERE {  
  GRAPH <http://example/bookStore> {  
    ?book dc:date ?date .  
    FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )  
    ?book ?p ?v  
  }  
}
```



Programming SPARQL Update in Jena



SPARQL Update in Jena

```
Dataset dataset = DatasetFactory.create();
```

```
UpdateAction.parseExecute("""  
+ "PREFIX ex: <http://example.org/myTDBTest#>"  
+ ""  
+ "INSERT DATA {"  
+ "    ex:Shakespeare ex:hasName 'Shakespeare' ."  
+ "}", dataset);
```

```
Model model = dataset.getDefaultModel();  
model.write(System.out, "TURTLE");
```



SPARQL queries



*The next slides are intended as “schemas”
to provide a quick overview of stuff
you already know something about...*

*The text book and links to W3C
documents
in the portal provide lots of examples!*



SELECT queries: Basic forms

- Basic form:
 - **SELECT** *projection* **WHERE** { *pattern* }
 - the *projection* is a list of *variables*
 - the *pattern* is (centrally) a list of *triples*
 - returns a table with *one row per result* and *one column per project variable*
- Optional and combinable variations:
 - SELECT ***** WHERE { ... }
 - SELECT **DISTINCT** ... WHERE { ... }
 - SELECT ... WHERE { ... } **LIMIT** *n*
 - SELECT ... WHERE { ... } **LIMIT** *n* **OFFSET** *m*
 - SELECT ... WHERE { ... } **ORDER BY** ...*variable*...
 - SELECT ... WHERE { ... } **ORDER BY** **DESC**(...*variable*...)



SELECT: Grouping

- Grouping of results:
 - SELECT *...grouping or aggregate variables...* WHERE { ... }
GROUP BY *...grouping variables...* [**HAVING** ...]
 - example: counting students in courses

```
SELECT COUNT(?student) WHERE {  
  ?student ex:takes ?course .  
}
```



SELECT: Grouping

- Grouping of results:
 - SELECT *...grouping or aggregate variables...* WHERE { ... }
GROUP BY *...grouping variables...* [**HAVING ...**]
 - example: counting students in courses

```
SELECT (COUNT(?student) AS ?count) WHERE {  
  ?student ex:takes ?course .  
}
```



SELECT: Grouping

- Grouping of results:
 - SELECT *...grouping or aggregate variables...* WHERE { ... }
GROUP BY *...grouping variables...* [**HAVING ...**]
 - example: counting students in courses

```
SELECT ?course (COUNT(?student) AS ?count) WHERE {  
    ?student ex:takes ?course .  
}  
GROUP BY ?course
```



SELECT: Grouping

- Grouping of results:
 - SELECT *...grouping or aggregate variables...* WHERE { ... }
GROUP BY *...grouping variables...* [**HAVING ...**]
 - example: counting students in courses

```
SELECT ?course (COUNT(?student) AS ?count) WHERE {  
    ?student ex:takes ?course .  
}  
GROUP BY ?course  
HAVING (?count >= 10)
```



SELECT: Grouping

- Grouping of results:
 - SELECT *...grouping or aggregate variables...* WHERE { ... }
GROUP BY *...grouping variables...* [**HAVING ...**]
 - example: counting students in courses

```
SELECT ?course (COUNT(?student) AS ?count) WHERE {  
    ?student ex:takes ?course .  
}
```

```
GROUP BY ?course  
HAVING (?count >= 10)  
ORDER BY DESC(?count)
```



SELECT: Grouping

- Grouping of results:
 - SELECT *...grouping or aggregate variables...* WHERE { ... }
GROUP BY *...grouping variables...* [**HAVING (...)**]
 - *grouping variables:*
 - regular variables
 - used to group the rows in the results
 - *aggregate variables:*
 - the results of aggregate functions:
SUM, COUNT, MIN, MAX, AVG, GROUP_CONCAT or
SAMPLE (aggregate functions)



SELECT: Bindings

- In the **list of projection variables**:
 - SELECT **(SUM(...) AS ...)** ... WHERE { ... }
 - SELECT **(...xsd:function...(...) AS ...)** ... WHERE { ... }
- In the graph **pattern**:
 - SELECT ... WHERE { ... **BIND(... AS ...)** ... }
- In the **GROUP BY** variable list
 - SELECT ... WHERE { ... }
GROUP BY ... **(... AS ...)** ...
- In-line values:
 - SELECT ... WHERE {
...
VALUES var { value1, value2, ... }
...
}



SELECT: Composite patterns

- WHERE-variants with multiple *pattern groups*:
 - ... WHERE { { ... } UNION { ... } }
 - ... WHERE { ... OPTIONAL { ... } }
 - ... WHERE { ... MINUS { ... } }
 - ... WHERE { ... FILTER (...) }
 - ... WHERE { ... FILTER [NOT] EXISTS { ... } }
- *Filters* are *logical expressions*:
 - standard logic operators: `!, &&, ||`
 - (in-)equality operators: `=, !=, <, <=, >, >=`
 - arithmetic: `+, -, /, *`
 - built-in, imported (*xsd:...*) and self-defined functions



SELECT: Built-in functions

- Examples of built-ins:
 - **bound**, **if**
 - **exists**, **not exists**
 - **in**, **not in**
 - **IRI**, **bnode**
 - **isIRI**, **isBlank**, **isLiteral**, **isNumeric**
 - **str**, **lang**, **strlang** (“for language tagged literals”@en)
 - **regex**, **strlen**, **contains**, **substr...** (from XPath)
 - **replace**
 - **abs**, **rand**, **ceil**, **floor**
 - **now**, **year**, **month**, **days**, **hours**, **minutes**, **seconds**



SELECT: Federated queries

- Local queries to non-default graphs:

```
SELECT ... WHERE { ... GRAPH <iri> { ... } . ... }
```

- Nested queries:

```
SELECT ... WHERE { ... { SELECT ... WHERE { ... } } }
```

- Remote queries:

```
SELECT ... WHERE { ... SERVICE <irl> { ... } }
```

```
SELECT ... WHERE { ... SERVICE SILENT <irl> { ... } }
```



SELECT queries: Naming graphs

- SELECT ... WHERE { ... }
 - matching a pattern from the default graph in the dataset
- SELECT ... WHERE { ... **GRAPH** <iri> { ... } }
 - matching a pattern from the *named graph* <iri> in the dataset
- SELECT ... **FROM** <irl1> WHERE { ... }
- SELECT ... **FROM** <irl1> ... <irlN> WHERE { ... }
 - the *union* of named graphs <irl1>...<irlN> becomes a *default graph* to be matched by the WHERE-pattern
- SELECT ... **FROM NAMED** <irl>
WHERE { ... **GRAPH** <irl> { ... } }
 - the *named graph* <irl> becomes a *named graph* to be explicitly matched by a part of the WHERE-pattern



SELECT: Composite properties

- *Property paths (in SPARQL 1.1):*
- Concatenation: a / b (means “first a , then a 's b ”)
- Repetition: a^* (0:n), a^+ (1:n), $a?$ (0:1)
- Alternative: $a | b$ (means “ a or b ”)
- Inversion: a (means “ a backwards”)
- Grouping: (\dots)
- Negation: $!a$ (means “any other predicate than a ”)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1):*
- Concatenation: ***a / b*** (means “first ***a***, then ***a***'s ***b***”)
 - example:
 ?***nephew*** :***hasParent*** ?***parent*** .
 ?***parent*** :***hasBrother*** ?***uncle*** .
 - can be written as:
 ?***nephew*** :***hasParent*** / :***hasBrother*** ?***uncle*** .
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means “***a*** or ***b***”)
- Inversion: ***^a*** (means “***a*** backwards”)
- Grouping: ***(...)***
- Negation: ***!a*** (means “any other predicate than ***a***”)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1):*
- Concatenation: ***a / b*** (means “first ***a***, then ***a***'s ***b***”)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
 - example:
 - ?person :hasParent* ?ancestor .
 - ?person :hasParent+ ?ancestor .
 - ?person :hasSibling? ?sibling .
- Alternative: ***a | b*** (means “***a*** or ***b***”)
- Inversion: ***^a*** (means “***a*** backwards”)
- Grouping: ***(...)***
- Negation: ***!a*** (means “any other predicate than ***a***”)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1):*
- Concatenation: ***a | b*** (means “first ***a***, then ***a***'s ***b***”)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
 - these two pattern-lines always match:
?anyResource :anyProperty* ?anyResource .
?anyResource :anyProperty? ?anyResource .
 - for any ***?anyResource***
 - and any ***:anyProperty***
- Alternative: ***a | b*** (means “***a*** or ***b***”)
- Inversion: ***^a*** (means “***a*** backwards”)
- Grouping: ***(...)***
- Negation: ***!a*** (means “any other predicate than ***a***”)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1):*
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means *a* or *b*)
 - example:
`?child :hasFather | :hasMother ?parent .`
`?person :hasBrother | :hasSister ?sibling .`
- Inversion: ***^a*** (means *a* backwards)
- Grouping: ***(...)***
- Negation: ***!a*** (any other predicate than *a*)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1):*
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means *a* or *b*)
- Inversion: ***^a*** (means *a* backwards)
 - example:
`?parent ^:hasParent ?child .`
- Grouping: ***(...)***
- Negation: ***!a*** (any other predicate than *a*)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1):*
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means *a* or *b*)
- Inversion: ***^a*** (means *a* backwards)
- Grouping: ***(...)***
 - example:
`?parent ^(:hasFather | :hasMother) ?child .`
- Negation: ***!a*** (any other predicate than *a*)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means *a* or *b*)
- Inversion: ***^a*** (means *a* backwards)
- Grouping: ***(...)***
 - example:
`?parent ^(:hasFather | :hasMother) ?child .`
`?parent (^:hasFather | ^:hasMother) ?child .`
- Negation: ***!a*** (any other predicate than *a*)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means *a* or *b*)
- Inversion: ***^a*** (means *a* backwards)
- Grouping: ***(...)***
 - example:
`?uncle ^(:hasParent / :hasBrother) ?nephew .`
`?uncle ?nephew .`
- Negation: ***!a*** (any other predicate than *a*)



SELECT: Composite properties

- *Property paths (in SPARQL 1.1)*:
- Concatenation: ***a / b*** (first *a*, then *a*'s *b*)
- Repetition: ***a**** (0:n), ***a+*** (1:n), ***a?*** (0:1)
- Alternative: ***a | b*** (means *a* or *b*)
- Inversion: ***^a*** (means *a* backwards)
- Grouping: ***(...)***
 - example:
`?uncle ^(:hasParent / :hasBrother) ?nephew .`
`?uncle (^:hasBrother / ^:hasParent) ?nephew .`
- Negation: ***!a*** (any other predicate than *a*)



The other query types

- ASK { ... }
 - *can the pattern be matched, yes or no?*
- CONSTRUCT { ?s1 ?p1 ?o1 . ?s2 ?p2 ?o2 } WHERE { ... }
 - *returns triples*, e.g., for copying a graph:
CONSTRUCT { ?s ?p ?o }
WHERE { GRAPH <iri> { ?s ?p ?o } . }
- DESCRIBE ?resource WHERE { ... }
 - *returns a “relevant excerpt” of the graph for ?resource*
 - not well defined: *all triples where a resource is subject?*
all triples where a resource is subject or object?
concise bounded descriptions (CBDs)?
symmetric CBDs?
- Most variations of SELECT can also be used for
ASK/CONSTRUCT/DESCRIBE when they give meaning!



Programming SPARQL in Jena



SELECT query

```
Model model = ModelFactory.createDefaultModel();
```

```
...
```

```
Query selectQuery = QueryFactory.create("""  
    + "SELECT DISTINCT ?type WHERE {"  
    + "  ?res <" + RDF.type + "> ?type ."  
    + "}");
```

```
QueryExecution queryExecution =
```

```
    QueryExecutionFactory.create(selectQuery, model);
```

```
ResultSet table = queryExecution.execSelect();
```

```
while (table.hasNext()) {
```

```
    QuerySolution row = table.nextSolution();
```

```
    System.out.println(row.get("type")); // returns RDFNode, or
```

```
    System.out.println(row.getResource("type")); // returns Resource
```

```
}
```

```
...
```

```
model.close();
```



SELECT query

```
Model model = ModelFactory.createDefaultModel();  
Dataset dataset = DatasetFactory.create(model);
```

...

```
Query selectQuery = QueryFactory.create("""  
    + "SELECT DISTINCT ?label WHERE {"  
    + "    ?res <" + RDFS.label + "> ?label ."  
    + "}"  
    );
```

```
QueryExecution queryExecution =  
    QueryExecutionFactory.create(selectQuery, dataset);  
ResultSet table = queryExecution.execSelect();  
while (table.hasNext()) {  
    QuerySolution row = table.nextSolution();  
    System.out.println(row.getLiteral("label")); // returns Literal  
}
```

...

```
dataset.close();
```



ASK query

```
Model model = ModelFactory.createDefaultModel();  
Dataset dataset = DatasetFactory.create(model);
```

```
...
```

```
Query askQuery = QueryFactory.create("""  
    + "ASK WHERE {"  
    + "    ?res <" + RDFS.label + "> ?label ."  
    + "}");
```

```
QueryExecution queryExecution =  
    QueryExecutionFactory.create(askQuery, dataset);  
Boolean answer = queryExecution.execAsk();
```

```
...
```

```
dataset.close()
```



CONSTRUCT query

```
Model model = ModelFactory.createDefaultModel();  
Dataset dataset = DatasetFactory.create(model);
```

...

```
String myOwnLabel = "http://example.org/label";  
Query constructQuery = QueryFactory.create("""  
    + "CONSTRUCT {"  
    + "    ?res <" + RDFS.label + "> ?label ."  
    + "} WHERE {"  
    + "    ?res <" + myOwnLabel + "> ?label ."  
    + "}");
```

```
QueryExecution queryExecution =  
    QueryExecutionFactory.create(constructQuery, dataset);  
Model resultModel = queryExecution.execConstruct();
```

...

```
dataset.close()
```



DESCRIBE query

```
Model model = ModelFactory.createDefaultModel();  
Dataset dataset = DatasetFactory.create(model);
```

```
...
```

```
String myOwnLabel = "http://example.org/label";  
Query describeQuery = QueryFactory.create("" +  
    + "DESCRIBE ?res WHERE {" +  
    + "  ?res <" + myOwnLabel + "> ?label ." +  
    + "}");
```

```
QueryExecution queryExecution =  
    QueryExecutionFactory.create(describeQuery, dataset);  
Model resultModel = queryExecution.execDescribe();
```

```
...
```

```
dataset.close()
```

