IN3060/4060 – Semantic Technologies – Spring 2021

Lecture 12: SPARQL 1.1

Jieying Chen

6th April 2021

Department of Informatics

University of Oslo

Today's Plan

1. SPARQL 1.1 QUERY language
   - Assignment and Expressions
   - Aggregates
   - Subqueries
   - Negation
   - Property paths
2. SPARQL 1.1 Federated Query
3. SPARQL 1.1 UPDATE Language
4. SPARQL 1.1 Entailment Regimes

SPARQL

- SPARQL Protocol And RDF Query Language
- Standard language to query graph data represented as **RDF triples**
- W3C Recommendations
  - **SPARQL 1.0:** W3C Recommendation 15 January 2008
  - **SPARQL 1.1:** W3C Recommendation 21 March 2013
- This lecture is about SPARQL 1.1.
- Documentation:
  - SPARQL 1.1 Query Language.
    `https://www.w3.org/TR/sparql11-query/`

Components of a SPARQL query

Prologue: prefix definitions Results form specification: (1) variable list, (2) type of query (SELECT, ASK, CONSTRUCT, DESCRIBE), (3) remove duplicates (DISTINCT, REDUCED) Dataset specification Query pattern: graph pattern to be matched Solution modifiers: ORDER BY, LIMIT, OFFSET

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX dbo:   <http://dbpedia.org/ontology/>
SELECT DISTINCT ?collab
FROM <http://dbpedia_dataset>
WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?pub dbo:starring ?jd .
    ?pub dbo:starring ?other .
    ?other foaf:name ?collab .
    FILTER (STR(?collab)!="Johnny Depp"@en)
}
```

Outline

SPARQL 1.1: new features

- The new features in SPARQL 1.1 QUERY language:
  1. Assignments and expressions
  2. Aggregates
  3. Subqueries
  4. Negation
  5. Property paths
  6. A short form for CONSTRUCT
  7. An expanded set of functions and operators
- SPARQL 1.1 UPDATE Language
- SPARQL 1.1 Federated Queries
- SPARQL 1.1 Entailment Regimes
- Rationale for the extensions of SPARQL 1.0
  `https://www.w3.org/TR/sparql-features/`

1. Assignment and Expressions

- The value of an expression can be assigned/bound to a new variable
- Can be used in SELECT, BIND or GROUP BY clauses: *(expression AS ?var)*

Books with price < 20 taking into account discount

```
SELECT ?title ?price WHERE
{
  ?x ns:price ?p .
  ?x ns:discount ?discount
  BIND (?p*(1-?discount) AS ?price)
  ?x dc:title ?title .
  FILTER(?price < 20)
}
```


1. Assignment and Expressions

- The value of an expression can be assigned/bound to a new variable
- Can be used in SELECT, BIND or GORUP BY clauses: *(expression AS ?var)*

Expressions in SELECT clause

```
SELECT ?title (?p AS ?fullPrice)
       (?fullPrice*(1-?discount) AS ?customerPrice) WHERE
{
  ?x ns:price ?p .
  ?x dc:title ?title .
  ?x ns:discount ?discount
}
```

2. Aggregates: Grouping and Filtering

- Aggregation (sum, count, etc.) works very much like in SQL
- Solutions can optionally be grouped according to one or more expressions.
- Aggregates (count, sum, etc.) are applied per group.
- To specify the group, use GROUP BY.
- If GROUP BY is not used, then only one (implicit) group
- To filter solutions resulting from grouping, use HAVING.
- HAVING operates over grouped solution sets, in the same way that FILTER operates over un-grouped ones.

2. Aggregates: Example

Counties of Norway with less than 15 municipalities

```
SELECT ?name (count(?kommune) AS ?kcount)
WHERE {
  ?county a gd:Fylke ;
          gn:officialName ?name ;
          gn:hasmunicipality ?kommune .
  ?kommune a gd:Kommune .
}
GROUP BY ?name
HAVING (?kcount < 15)
```

Note: Only expressions consisting of aggregates and constants may be projected, together with variables in GROUP BY.

2. Aggregates: functions

- Count counts the number of times a variable has been bound.
- Sum sums numerical values of bound variables.
- Avg finds the average of numerical values of bound variables.
- Min finds the minimum of the numerical values of bound variables.
- Max finds the maximum of the numerical values of bound variables.
- Group_Concat creates a string with the values concatenated, separated by some optional character.
- Sample just returns a sample of the values.

3. Subqueries

- Subqueries are a way to embed SPARQL queries within other queries
- To achieve results which cannot otherwise be achieved, e.g. computing intermediate values in a subquery

Return the largest city in each country

```
SELECT ?ctry ?city WHERE {
  {SELECT ?ctry (MAX(?cityPop) AS ?maxCityPop) WHERE {
     ?city :cityInCountry ?ctry; :hasPop ?cityPop} GROUP BY ?ctry}
  ?city :cityInCountry ?ctry; :hasPop ?maxCityPop.
}
```

- Subqueries are evaluated logically first, and the results bind variables in the outer query.
- Only variables selected in the subquery will be visible, or in scope, to the outer query.

4. Negation in SPARQL 1.0

Remember: No negation in SPARQL 1.0 because of Monotonicity
Well actually...

People without names

```
SELECT DISTINCT * WHERE {
    ?person a foaf:Person .
    OPTIONAL {
        ?person foaf:name ?name .
     FILTER (!bound(?name))
     }
}
```

The BOUND function provides a loophole.
However, this is not very easy to write.

4. Negation in SPARQL 1.1

Two ways to do negation: MINUS and FILTER NOT EXISTS

People without names, using MINUS

```
SELECT DISTINCT * WHERE {
    ?person a foaf:Person .
    MINUS { ?person foaf:name ?name }
}
```

- A MINUS B evaluates both A and B giving solutions $sol(A)$ and $sol(B)$
- The solutions of A MINUS B are all $s_a \in sol(A)$ except the ones where there is a $s_b \in sol(B)$ with
    - s_A and s_B compatible, and
    - s_A and s_B have some bound variables in common

4. Negation in SPARQL 1.1 (cont.)

People without names, using FILTER NOT EXISTS

```
SELECT DISTINCT * WHERE {
    ?person a foaf:Person .
    FILTER NOT EXISTS { ?person foaf:name ?name }
}
```

- A FILTER NOT EXISTS B evaluates A and for each solution $s_A \in sol(A)$ it checks...
- ...given the bindings from s_A...
- ...if there is a match for B...
- ...and discards s_A if there is.

4. Negation in SPARQL 1.1 (cont.)

They may produce different results. Data with `ex:Ernesto a foaf:Person`

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

Does not remove solutions (no shared variables!) and returns `ex:Ernesto a foaf:Person`

```
SELECT DISTINCT * WHERE {
    ?s ?p ?o .
    FILTER NOT EXISTS { ?x ?y ?z }
}
```

Returns no solutions. Since there are not shared variables, it removes all solutions.

Open and Closed World Assumptions

Aggregates and negation assume Closed World and Unique names!
The answers are only true with respect to the current dataset.

- "As far as we know, there are 13 municipalities in Vestfold."
- Can't say: "they don't have names", can say: "we don't know their names".
- "As far as we know, no-one has climbed that mountain."
- "Based on the available data, the average fuel price is currently 13.37 NOK/l."

This will have implications when combined with reasoning.

5. Property paths: basic motivation

- Some queries get needlessly complex.
- "property paths" can take the place of the predicate in graph patterns
- E.g. write `?x foaf:maker|dct:creator ?p` instead of using `UNION`.
- To get friend's name, go `{ _:me foaf:knows/foaf:name ?friendsname }`.
- Sum several items:
  `SELECT (sum(?cost) AS ?total) { :order :hasItem/:price ?cost }`
- etc.
- Adds a small property-oriented query language inside the language.

5. Property paths: syntax

Syntax Form	Matches
`iri`	An (property) IRI. A path of length one.
`^elt`	Inverse path (object to subject).
`elt1 / elt2`	A sequence path of `elt1` followed by `elt2`.
`elt1 | elt2`	A alternative path of `elt1` or `elt2` (all possibilities are tried).
`elt*`	Seq. of zero or more matches of `elt`.
`elt+`	Seq. of one or more matches of `elt`.
`elt?`	Zero or one matches of `elt`.
`!iri` or `!(`iri_1`| ...|`iri_n`)`	Negated property set.
`!^iri` or `!(^`iri_i`| ...|^`iri_n`)`	Negation of inverse path.
`!(`iri_1`|...|`iri_j`|^`iri_{j+1}`|...|^`iri_n`)`	Negated combination of forward and inverese properties.
`(elt)`	A group path `elt`, brackets control precedence.

* `elt` is a path element, which may itself be composed of path constructs (see Syntax form).

5. Property paths: example

The names of all friends of Ernesto's friends

```
SELECT ?name WHERE {
  uio:Ernesto foaf:knows+ ?friend
  ?friend foaf:name|foaf:givenName ?name .
}
```

Outline

Federated query support

- The `SERVICE` keyword instructs a federated query processor to invoke a portion of a SPARQL query against a remote SPARQL service/endpoint.
- SPARQL service: any implementation conforming to the *SPARQL 1.1 Protocol for RDF*

Combining local file with remote SPARQL service

```
SELECT ?name
FROM <http://example.org/mylocalfoaf.rdf>
WHERE {
  <http://example.org/mylocalfoaf/I> foaf:knows ?person .
  SERVICE <http://people.example.org/sparql> {
    ?person foaf:name ?name .    }
}
```


Outline

SPARQL 1.1 UPDATE

- Do not confuse with CONSTRUCT
- CONSTRUCT is an alternative for SELECT
- Instead of returning a table of result values, CONSTRUCT returns an RDF graph according to the template
- SPARQL 1.1 UPDATE is a language to modify the given GRAPH
- `https://www.w3.org/TR/2013/REC-sparql11-update-20130321/`

SPARQL 1.1 UPDATE: Inserting and deleting triples

Inserting triples in a graph

```
INSERT DATA {
  GRAPH </graph/courses/> {
    <course/in3060> ex:taughtBy <staff/jieyingc> .
    <staff/jieyingc> foaf:name "Jieying Chen" ;
} }
```

Deleting triples from a graph

```
DELETE DATA {
  GRAPH </graph/courses/> {
    <course/in3060> ex:oblig <exercise/oblig6> .
    <exercise/oblig6> rdfs:label "Mandatory Exercise 6" .
} }
```

If no `GRAPH` is given, default graph is used.

SPARQL 1.1 UPDATE: Inserting conditionally

Most useful when inserting statements that you already have, but hold true for something else.

Inserting triples for another subject

```
INSERT {
  <http:// .../geo/inndeling/03> a gd:Fylke ;
       gn:name "Oslo" ;
       ?p ?o .
}
WHERE {
  <http:// .../geo/inndeling/03/0301> a gd:Kommune ;
            ?p ?o .
}
```


SPARQL 1.1 UPDATE: Deleting conditionally

From specification:

Deleting old books

```
DELETE {
   ?book ?p ?v .
}
WHERE {
  ?book dc:date ?date .
  FILTER ( ?date < "2000-01-01T00:00:00"^^xsd:dateTime )
  ?book ?p ?v .
}
```


SPARQL 1.1 UPDATE: Deleting conditionally, common shortform

Deleting exactly what's matched by the `WHERE` clause.

Deleting information about the course inf3580

```
DELETE WHERE {
  ?s ?p <http://ifi.uio.no/courses/inf3580> .
}
```

SPARQL 1.1 UPDATE: Delete/Insert full syntax

In most cases, you would delete some triples first, then add new, possibly in the same or other graphs.
From specification:

All the possibilities offered by DELETE/INSERT

```
( WITH IRIref )?
( ( ( DELETE QuadPattern ) ( INSERT QuadPattern )?  )  | (INSERT
QuadPattern) )
( USING ( NAMED )? IRIref )*
WHERE GroupGraphPattern
```


SPARQL 1.1 UPDATE: Delete/Insert simple example

Update user information

```
DELETE {
  <http:// .../user/larshvermannsen> ?p ?o .
}
INSERT {
  <http:// .../user/larshvermannsen> a sioc:User ;
    rdfs:label """Lars Hvermannsen"""@no ;
    sioc:email <mailto:lars@hvermannsen.no> ;
    sioc:has_function <http:// .../role/Administrator> ;
    wdr:describedBy status:inaktiv .
}
WHERE {
  <http:// .../user/larshvermannsen> ?p ?o .
}
```


SPARQL 1.1 UPDATE: Delete/Insert example with named graphs

Update user information

```
DELETE {
  GRAPH </graphs/users/> {
    <http:// .../user/larshvermannsen> ?p ?o .
  }
}
INSERT {
  GRAPH </graphs/users/> {
    <http:// .../user/larshvermannsen> a sioc:User ;
          rdfs:label """Lars Hvermannsen"""@no .
  }
}
USING </graphs/users/> WHERE {
    <http:// .../user/larshvermannsenno> ?p ?o .
}
```


SPARQL 1.1 UPDATE: Delete/Insert example explained

- USING plays the same role as FROM.
- GRAPH says where to insert or delete.
- This makes it possible to delete, insert and match against different graphs.

SPARQL 1.1 UPDATE: Delete/Insert example with single named graphs

Update user information

```
WITH </graphs/users/>
DELETE {
  <http:// .../user/larshvermannsen> ?p ?o .
}
INSERT {
  <http:// .../user/larshvermannsen> a sioc:User ;
        rdfs:label """Lars Hvermannsen"""@no .
}
WHERE {
  <http:// .../user/larshvermannsenno> ?p ?o .
}
```

Equivalent to the previous query!

SPARQL 1.1 UPDATE: Whole graph operations

From the specification:

`LOAD ( SILENT )? IRIref_from ( INTO GRAPH IRIref_to )?`
: Loads the graph at IRIref_from into the specified graph, or the default graph if not given.

`CLEAR ( SILENT )? (GRAPH IRIref | DEFAULT | NAMED | ALL )`
: Removes the triples from the specified graph, the default graph, all named graphs or all graphs respectively. Some implementations may remove the whole graph.

`CREATE ( SILENT )? GRAPH IRIref`
: Creates a new graph in stores that record empty graphs.

`DROP ( SILENT )? (GRAPH IRIref | DEFAULT | NAMED | ALL )`
: Removes the specified graph, the default graph, all named graps or all graphs respectively. It also removes all triples of those graphs.

Also provides shortcuts, `COPY`, `MOVE` and `ADD`.
Usually, `LOAD` and `DROP` are what you want.

Outline

Entailment regimes: overview

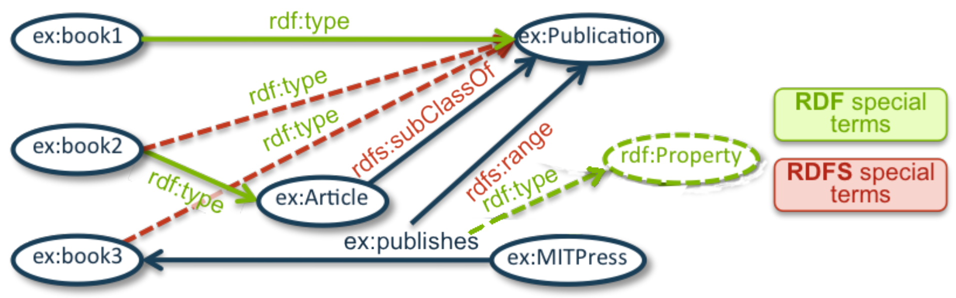
- Gives guidance for SPARQL query engines
- Basic graph pattern by means of subgraph matching: *simple entailment*
- Solutions that implicitly follow from the queried graph: *entailment regimes*
- **RDF entailment**, **RDF Schema entailment**, D-Entailment, **OWL 2 RDF-Based Semantics entailment**, **OWL 2 Direct Semantics entailment**, and RIF-Simple entailment
- `https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/`

Entailment regimes: example (1)

- ex:book1 rdf:type ex:Publication .
- ex:book2 rdf:type ex:Article .
- ex:Article rdfs:subClassOf ex:Publication .
- ex:publishes rdfs:range ex:Publication .
- ex:MITPress ex:publishes ex:book3 .

QUERY 1: SELECT ?prop WHERE ?prop rdf:type rdf:Property
QUERY 2: SELECT ?pub WHERE ?pub rdf:type ex:Publication

Entailment regimes: example (2)



Dashed lines: inferred triples

Entailment regimes: example (3)

- ex:book1 rdf:type ex:Publication .
- ex:book2 rdf:type ex:Article .
- ex:Article rdfs:subClassOf ex:Publication .
- ex:publishes rdfs:range ex:Publication .
- ex:MITPress ex:publishes ex:book3 .

Query 1: Using RDF entailment regime (new entailed triples):

- ex:publishes rdf:type rdf:Property .

Query 2: Using RDFS entailment regime (new entailed triples):

- ex:book2 rdf:type ex:Publication .
- ex:book3 rdf:type ex:Publication .

(Graph matching is performed over the extended RDF graph)

The OWL Entailment Regimes

- OWL 2 RDF-based Semantics Entailment Regime
- OWL 2 Direct Semantics Entailment Regime

- https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/
- Birte Glimm. **Using SPARQL with RDFS and OWL entailment**. International Conference on Reasoning Web, 2011

OWL 2 Direct Semantics Entailment Regime

- OWL 2 Direct Semantics is our DL-semantics
  - Separates classes, properties, individuals, etc.
  - Classes interpreted as sets, Properties as relations
- Direct Semantics Entailment Regime works on restricted RDF graphs and Queries

Technical solution: Variable Typing

- Require a type on every variable in a query
- SELECT . . . FROM { . . . ?x rdf:type TYPE . . . . }
- Where TYPE can a class or *one* of: owl:Class, owl:ObjectProperty, owl:DatatypeProperty, owl:Datatype, or owl:NamedIndividual

OWL 2 RDF-based Semantics Entailment Regime

- RDF-based semantics is the one with two steps in Oblig 5
  - I_S interprets class and relation URIs as domain elements,
  - I_{EXT} maps these to relations the domain
  - Not every relation on domain is I_{EXT} of something
- No need for mapping an RDF graph into OWL objects
- This may lead to less consequences than expected (Incompleteness)

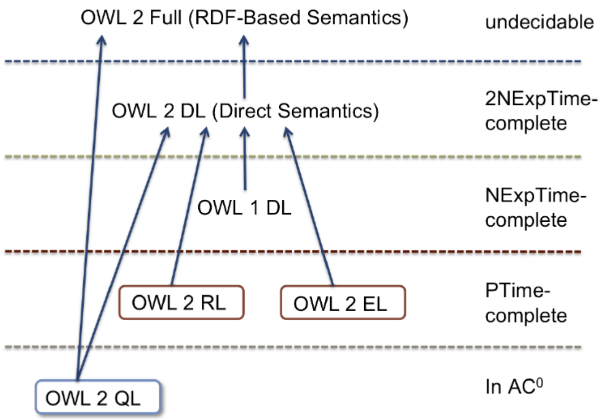
OWL 2 Entailment Regimes: example

- Graph: `ex:a rdf:type ex:C`
- BGP in query:

```
?x rdf:type
  [
  rdf:type owl:Class ;
  owl:unionOf( ex:C ex:D )
  ]
```

- OWL/RDF for: $(C \sqcup D)(x)$
- `ex:a` not returned in the solution for `?x` using OWL 2 RDF-Based Semantics
  - G does not include that this union is the class extension of any domain element
  - Solution: add statement `ex:E owl:unionOf ( ex:C ex:D )`
  - This type of statement may lead to undecidability
- `ex:a` would be a solution for `?x` using OWL 2 Direct Semantics
  - classes denote sets and not domain elements

OWL 2 Entailment Regimes: Complexity and Profiles



OWL 2 Entailment Regimes: Systems

- **OWL-BGP:** SPARQL implementation where basic graph patterns are evaluated with OWL 2 Direct Semantics.
  - `https://github.com/iliannakollia/owl-bgp`
- **RDFox:** highly scalable in-memory RDF triple store that supports parallel datalog reasoning.
  - OWL 2 RL axioms can be directly transformed to datalog rules
  - `https://www.cs.ox.ac.uk/isg/tools/RDFox/`
- **ontop:** answering SPARQL queries over databases under OWL 2 QL Entailment regime
  - Ontop is a platform to query relational databases as Virtual RDF Graphs using SPARQL
  - An Ontology in OWL 2 QL and R2RML mappings
  - R2RML: RDB to RDF Mapping Language
  - `http://ontop.inf.unibz.it/`