Semantic Web Technologies
SPARQL

Heiko Paulheim
Previously on “Semantic Web Technologies”

- We have got to know
  - The RDF and RDFS languages
  - The Linked Open Data paradigm

- We have accessed Linked Open Data
  - with browsers and via Jena
  - jumping from node to node in the graph

- ...let us have a closer look!
An Example RDF Graph

Germany

lies in
lies in
lies in
lies in
lies in

... 

0001

0002

0003

0004

0020

0021

0022

inhabitants

inhabitants

inhabitants

inhabitants

inhabitants

3.501.872

691.518

1.378.176

1.798.836

state

state

state

state

Question: in which states are the five biggest cities of Germany located?
Information Retrieval on Linked Open Data

• Question: in which states are the five biggest cities of Germany located?

• So let's try...
Conclusion

• This is very tedious!
• Are there any simpler mechanisms available?
Semantic Web – Architecture

Semantic Web Technologies (This lecture)

here be dragons...

Technical Foundations

Query: SPARQL
Ontology: OWL
Rules: RIF
Schema: RDF-S
Data Interchange: RDF
Data Interchange: XML
URI
Unicode

User Interface and Applications
Proof
Unifying Logic
Trust
Cryptography

Berners-Lee (2009): Semantic Web and Linked Data
What Would We Like to Have?

Question: in which states are the five biggest cities of Germany located?

Germany

lies in

lies in

lies in

lies in

lies in

...
Wanted: A Query Language for the Semantic Web

- ...just like SQL is for relational databases

```
SELECT  name, birthdate FROM customers
WHERE   id = '00423789'
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>birthdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>00183283</td>
<td>Stephen Smith</td>
<td>23.08.1975</td>
</tr>
<tr>
<td>00423782</td>
<td>Julia Meyer</td>
<td>05.09.1982</td>
</tr>
<tr>
<td>00789534</td>
<td>Sam Shepherd</td>
<td>31.03.1953</td>
</tr>
<tr>
<td>00423789</td>
<td>Herbert King</td>
<td>02.04.1960</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Wanted: A Query Language for the Semantic Web

- **SPARQL**: "SPARQL Query Language for RDF"
  - a recursive acronym

- A W3C Standard since 2008
- Allows for querying RDF graphs
Hello SPARQL!

• Example:

```
SELECT ?child
WHERE { :Stephen :fatherOf ?child }
```

Expressions with ? denote variables
SPARQL Basics

• Basic structure:

  SELECT <list of variables>
  WHERE { <pattern> }

• Variables denoted with ?

• Prefixes as in RDF/N3:

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

• The `<pattern>` in the WHERE clause is like N3
  – with variables

• `{?p foaf:name ?n }`
• `{?p foaf:name ?n; foaf:homepage ?hp }`
• `{?p foaf:knows ?p1, ?p2 }`
SPARQL: Pattern Matching on RDF Graphs

• WHERE clause: an RDF graph with variables

```
SELECT ?person1 ?person2
WHERE {
  ?person1 :knows \?otherPerson .
}
```

• Result:
  – \?person1 = :Peter, \?person2 = :Julia
SPARQL: Pattern Matching on RDF Graphs

• A person who has a daughter and a son:
  { ?p :hasDaughter ?d ; :hasSon ?s . }

• A person knowing two persons who know each other

• A person who has two children:
  { ?p :hasChild ?c1, ?c2 . }
• A person who has two children:
  \{ \text{?p :hasChild ?c1, ?c2 .} \}

• ResultSet:
  – \text{?p=:Stephen, ?c1=:Julia, ?c2=:Julia}

Observation: different variables are not necessarily bound to different resources!
SPARQL: Blank Nodes

- **WHERE clause**: an RDF graph with variables

  ```sparql
  SELECT ?person1 ?person2 ?otherPerson
  WHERE {
    ?person1 :knows ?otherPerson .
  }
  ```

- **Result**:
  - ?person1 = :Peter, ?person2 = :Julia; ?otherPerson = _:x1

- **Note**: Blank Node IDs are only unique within one result set!
SPARQL: Matching Literals

- **Strings**
  
  \[
  \{ \text{?country :name "Germany" . } \}
  \]

- **Watch out for language tags!**
  
  \[
  \{ \text{?country :name "Germany"@en . } \}
  \]

  \[→ \text{The Strings "Germany" and "Germany"@en are different!} \]

- **Numbers:**
  
  \[
  \{ \text{?person :age "42"^^xsd:int . } \}
  \]

  **Short hand notation:**
  
  \[
  \{ \text{?person :age 42 . } \}
  \]
SPARQL: Filters

• Used for further restricting results
  
  ```{?person :age ?age . FILTER(?age < 42) }```

• Operators for comparisons:
  
  `=  !=  <  >  <=  >=`

• Logical operations:
  
  `&&  ||  !`
SPARQL: Filters

• Persons with younger siblings

   FILTER(?a2 < ?a1)\}

• Persons that have both younger and older siblings

   FILTER(?a2 < ?a1 && ?a3 > ?a1)\}
SPARQL: Filters

- Second try: a person with two children
  
  $$\{ \ ?p :\text{hasChild} \ ?c1, \ ?c2 . \ \text{FILTER}( \ ?c1 \ neq \ ?c2) \}$$

- A slight improvement
  
  → Variables are now bound to different resources

- But: we still have the Non-Unique Naming Assumption
  
  → i.e., given that
  
  :Peter :hasChild :Julia .
  :Peter :hasChild :Stefan .

  we still cannot conclude that Peter has two children!

- Furthermore, there is still the Open World Assumption
  
  → i.e., Peter could also have more children
Filters for Strings

• Searching in Strings: using regular expressions

• People called “Ann”

  `{?person :name ?n . FILTER(regex(?n,"^Ann$")) }`
  `{?person :name ?n . FILTER(regex(?n,"Ann")) }`
  → the second variant would also find, e.g., “Mary-Ann”

• `str`: URIs and Literals as strings
• allows for, e.g., searching for literals across languages

  `{?country :name ?n . FILTER(str(?n) = "Tyskland") }`
Further Built-In Features

• Querying the type of a resource
  – isURI
  – isBLANK
  – isLITERAL

• Querying for the data type and language tags of literals
  – DATATYPE(?v)
  – LANG(?v)

• Comparing the language of two literals
  – langMATCHES(?v1, ?v2)
  – Caution: given ?v1 = "Januar"@DE, ?v2 = "Jänner"@DE—at
    LANG(?v1) = LANG(?v2) → false
    langMATCHES(?v1, ?v2) → true
Combining Patterns

• Find the private and work phone number

{ ?p :privatePhone ?nr }
UNION { ?p :workPhone ?nr }

• UNION creates a set union

?p = :peter, ?nr = 123;
?p = :john, ?nr = 234;
?p = :john, ?nr = 345;
...

That happens if John has both a private and a work phone
Optional Patterns

• Finde a person's phone number and fax number, **if existing**

```
{ ?p :phone ?tel }
OPTIONAL { ?p :fax ?fax }
```

• **OPTIONAL** also creates unbound variables

```
?p = :julia, ?nr = 978; ?fax = 349;
...  
```

**Unbound variable:** John does not have a fax number (as far as we know)
Unbound Variables

• Variables can remain unbound
• We can test this with BOUND

• Everybody who has a phone or a fax (or both):

  OPTIONAL { ?p :phone ?tel . }  
  OPTIONAL { ?p :fax ?fax . }  
  FILTER ( BOUND(?tel) || BOUND(?fax) )
Negation

• This is a common question w.r.t. SPARQL

• How do I do this:
  – "Find all persons who do not have siblings."

• This is left out of SPARQL intentionally!

• Why?

• Open World Assumption
  – we cannot know!

• For the same reason, there is no COUNT
  – at least not in standard SPARQL
Negation – Hacking SPARQL

• However, there is a possibility
  – try with caution!

• Using **OPTIONAL** and **BOUND**

• Find all persons without siblings

  ```sparql
  OPTIONAL {?p :hasSibling ?s . }
  FILTER ( !BOUND(?s) )
  ```

• This works

• However, you should know what you are doing
  – ...and how to interpret the results!
Negation – Hacking SPARQL

• How does that work?

• Results before FILTER:

  OPTIONAL { ?p :hasSibling ?s . }

  ?p = :peter, ?s = :julia
  ?p = :julia, ?s = :peter
  ?p = :mary, ?s =
  ?p = :paul, ?s =

• Applying the FILTER

  - FILTER(!BOUND(?s))

  ?p = :mary, ?s =
  ?p = :paul, ?s =

Unbound variables
Sorting and Paging Results

• **Sorting:** `ORDER BY ?name`

• **Limitations:** `LIMIT 100`

• **Lower Bounds:** `OFFSET 200`

• **Example:** persons 101-200, ordered by name
  – `ORDER BY ?name LIMIT 100 OFFSET 100`

• **LIMIT/OFFSET without ORDER BY:**
  – Result orderings are not deterministic
  – There is no default ordering
Sorting and Paging Results

• Application scenarios:
  – Some SPARQL services limit their result set sizes
  – Pre-loading in applications

• Application example:
  – let the user browse cities
  – it is more likely that users want to see the big cities
  – display 100 biggest cities on one page, show more on demand

• SELECT ?city ?population
  WHERE {?city hasPopulation ?population}
  ORDER BY DESC(?population)
  LIMIT 100
Filtering Duplicates

- SELECT DISTINCT ?person
  WHERE { ?person :privatePhone ?nr }
  UNION { ?person :workPhone ?nr }

- This means: all results with identical variable bindings are filtered

- This does not mean: persons identified by ?person are actually different

- Why?
  - Non-unique naming assumption
Custom Built-Ins

- Some SPARQL engines allow special constructs
- also known as *Custom Built-Ins*
- Example: geographic processing
  - Dataset: Linked Geo Data
LinkedGeoData

• A LOD Wrapper for OpenStreetMaps
• Querying for coordinates
  – simple:
    WHERE { ?x geo:long ?long; geo:lat ?lat .
    FILTER (?long>8.653 && ?long>8.654 &&
           ?lat>49.878 && ?lat>49.879)}

• More complex queries
  – all cafés within a 1km radius of a given point
    WHERE { ?x rdf:type lgdo:Cafe; geo:geometry ?geo .
    FILTER (bif:st_intersects(
           ?geo, bif:st_point(8.653, 49.878), 1))}
Custom Built-Ins

• Even more complex queries
  – all cafés within a 1km radius from a university

WHERE { ?x rdf:type lgdo:Cafe;
    geo:geometry ?cafegeo .
  ?y rdf:type lgdo:University;
    geo:geometry ?ugeo .
  FILTER (bif:st_intersects(?cafegeo, ?ugeo, 1))}
Further Query Types: ASK

- So far, we have only looked at SELECT queries
- ASK allows for yes/no queries:
  - e.g., are there persons with siblings?

  \[
  \text{ASK} \ { ?p \ :\text{hasSibling} \ ?s \ . } \]

- Often faster than SELECT queries

- The answer is true or false
  - \textit{false} means: there are no matching sub graphs
  - do not misinterpret (Open World Assumption!)
Further Query Types: DESCRIBE

• All properties of a resource
  
  DESCRIBE <http://dbpedia.org/resource/Berlin>

• Can be combined with a WHERE clause
  
  DESCRIBE ?city WHERE { :Peter :livesIn ?city . }

• Allows for exploration of a dataset with unknown structure

• Caution: types of results are not standardized, results vary from implementation to implementation!
Further Query Types: CONSTRUCT

• Creates a new RDF graph

CONSTRUCT

{ ?x rdfs:seeAlso
  <http://dbpedia.org/resource/Berlin> . }
WHERE
  FILTER (isURI(?x)) }

• CONSTRUCT returns complete RDF graphs
  – e.g., for further processing
SPARQL: Wrap-Up

- SPARQL is a query language for the semantic web
- Basic principle: pattern matching on graphs
- SPARQL allows for directed search for information instead of navigating the graph from node to node

- Results follow the semantic principles of RDF!
  - Open World Assumption
  - Non-unique naming assumption
Example: Jena + SPARQL

• Querying models with SPARQL

```java
String queryString = "SELECT ?x ...";
Query query = QueryFactory.create(queryString);
QueryExecution qe =
    QueryExecutionFactory.create(query, model);
ResultSet results = qe.execSelect();
while(results.hasNext()) {
    QuerySolution sol = results.next();
    String s = sol.get("x").toString();
    ...
}
```
Recap: Reasoning with Jena

• Given: a schema and some data

    Model schemaModel = ModelFactory.createDefaultModel();
    InputStream IS = new FileInputStream("data/example_schema.rdf");
    schemaModel.read(IS);

    Model dataModel = ModelFactory.createDefaultModel();
    IS = new FileInputStream("data/example_data.rdf");
    dataModel.read(IS);

    Model reasoningModel =
        ModelFactory.createRDFSModel(schemaModel, dataModel);

• Now, reasoningModel contains all derived facts
Example: Jena + SPARQL + Reasoning

• Derived facts can also be queries with SPARQL

• Given the `reasoningModel`

  ```java
  Query q = QueryFactory.create(
      "SELECT ?t WHERE
      { <http://example.org/Madrid>
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
          ?t .}");
  QueryExecution qexec =
      QueryExecutionFactory.create(q, reasoningModel);
  ResultSet rs = qexec.execSelect();
  while (rs.hasNext())
    String type = rs.next().get("t");
  ```

• Here, the query produces two solutions
  
  - http://example.org/City
  - http://www.w3.org/2000/01/rdf-schema#Resource
Accessing Public SPARQL Endpoints

• SPARQL Endpoints are an important building block of the Semantic Web tool stack

• Access using Jena:

```java
String query = "SELECT ...";
String endpoint = "http://dbpedia.org/sparql";
Query q = QueryFactory.create(strQuery);
QueryExecution qexec =
    QueryExecutionFactory.sparqlService(endpoint, q);
ResultSet RS = qexec.executeSelect();
```
Accessing Public SPARQL Endpoints

• Recap:
  – Jena uses the iterator pattern quite frequently

• Observation:
  – SPARQL ResultSets are also like iterators
  – Data can be retrieved from the server little by little
Semantic Web – Architecture

here be dragons...

Semantic Web Technologies (This lecture)

Technical Foundations

Berners-Lee (2009): Semantic Web and Linked Data
Questions?