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 programming frameworks
  – jumping from node to node in the graph

• ...let us have a closer look!
Question: in which states are the five biggest cities of Germany located?
Question: in which states are the five biggest cities of Germany located?

So let's try...
Information Retrieval on Linked Open Data

• Observations
  – Navigation across derefencable URIs ultimately leads to a goal
  – But it is tedious
  – A lot of useless data is potentially retrieved

• Different information needs
  – Good for simple factual questions
  – Less efficient for more complex questions
Semantic Web – Architecture

here be dragons...

Semantic Web Technologies (This lecture)

Technical Foundations

User Interface and Applications

Trust

Proof

Unifying Logic

Query: SPARQL

Ontology: OWL

Rules: RIF

Schema: RDF-S

Data Interchange: RDF

Data Interchange: XML

URI

Unicode

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

0001

state

inhabitants 3.501.872

0002

state

inhabitants 691.518

0003

state

inhabitants 1.378.176

0004

state

inhabitants 1.798.836

...
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>
</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:

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

• Variables denoted with ?

• Prefixes as in RDF/N3:

```sql
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 Basics: Graph Pattern Matching**

- **Pattern**: ?x :knows :Julia .
  - **Result**: \{ ?x = :Peter \}
SPARQL Basics: Graph Pattern Matching

- Pattern: :Julia :knows ?x .
  - Result: { ?x = :Stephen, ?x = :Ellen }

Diagram:
- :Peter :knows :Julia
- :Julia :knows :Stephen
- :Julia :knows :motherOf :Ellen
- :Julia :knows :fatherOf :Stephen
SPARQL Basics: Graph Pattern Matching

  - Result: { (?x = :Julia, ?y = :Stephen) ; (?x = :Julia, ?y = :Ellen)}
SPARQL: Pattern Matching on RDF Graphs

- A person who has a daughter and a son:
  \[
  \{ \text{?p :hasDaughter ?d ; :hasSon ?s .} \}
  \]

- A person knowing two persons who know each other
  \[
  \]

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

\{ \textcolor{red}{?p} :\text{hasChild} \ ?c1, \ ?c2 . \} \\

ResultSet:

- \textcolor{red}{?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

\[
\text{SELECT } \text{?person1 } \text{?person2 } \text{?otherPerson} \\
\text{WHERE } \{ \text{?person1 :knows } \text{?otherPerson} . \text{?otherPerson :fatherOf } \text{?person2} . \} \\
\]

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

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

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

- **Watch out for language tags!**
  
  \[
  \{ \ ?country :name "Germany"@en . \ }
  \]
  
  → The Strings "Germany" and "Germany"@en are different!

- **Numbers:**
  
  \[
  \{ \ ?person :age "42"^^xsd:int . \ }
  \]
  
  Short hand notation:
  
  \[
  \{ \ ?person :age 42 . \ }
  \]
SPARQL: Filters

• Used for further restricting results

\{ ?person :age ?age . FILTER(?age < 42) \}

• Operators for comparisons:

=  !=  <  >  <=  >=

• Logical operations:

&&  |  |  !
**SPARQL: Filters**

- **Persons with younger siblings**
  
  ```sparql
    FILTER(?a2 < ?a1)}
  ```

- **Persons that have both younger and older siblings**
  
  ```sparql
    FILTER(?a2 < ?a1 && ?a3 > ?a1)}
  ```

**Question:** Why do we get different persons for p2 and p3?
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”

```sparql
{?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

```sparql
{?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-
    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
Interlude: A Real-World Example


Der SPIEGEL, 27/2016, p. 52
Interlude: A Real-World Example

Who is this Walter K.?

Interlude: A Real-World Example

Who is this Walter K.?

SELECT DISTINCT(?x) WHERE {
    ?x a dbo:SoccerPlayer .
    ?x dbo:careerStation ?s. ?s dbo:team dbr:Germany_national_football_team.
}
Interlude: A Real-World Example

Who is this Walter K.?


We get one result:

<http://dbpedia.org/resource/Walter_Kelsch>
Auf schief Bahn geraten
Ex-Nationalspieler Kelsch sitzt in U-Haft


Der ehemalige Fußball-Nationalspieler Walter Kelsch sitzt wegen Drogenhandels im Internet in Stuttgart-Stammheim in Untersuchungshaft. Dies bestätigte die Staatsanwaltschaft im niedersächsischen Verden.


Man verwehre den Tat verdächtigen vor, "als Gruppierung 'Chemical-Love' über ein vorwiegend deutschsprachiges Dark-Market Forum sowie über einen eigenen Webshop Kokain und diverse synthetische Drogen vertrieben zu haben", hieß es in einer Pressemeldung. Insgesamt seien 54 kg Amphetamin, etwa 4 kg Heroin, rund 1,3 kg Kokain und etwa 25.000 Ecstasy-Tabletten sichergestellt worden.


Quelle: ndr.de, whalsid
Optional Patterns

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

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

• **OPTIONAL** also creates unbound variables

```query
?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):

```sparql
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 =
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
Custom Built-Ins

• 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)))}
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?
    
    \[
    \text{ASK } \{ \text{?p :hasSibling ?s . } \}
    \]
- 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
Query Federation

- Queries can be answered over *multiple* SPARQL endpoints
- Example

```
SELECT ?name ?lat ?long WHERE {
    ?x a foaf:Person .
    ?x foaf:name ?name .
    ?city owl:sameAs ?geocity .

    SERVICE <http://linkedcoordinates.org/sparql> {
        ?geocity geo:lat ?lat .
    }
}
```

```
:John

foaf:name

:London

foaf:based_near

“John Smith”

:534789

owl:sameAs

“51.50939”

geo:lat

“-0.11832”

geo:long
```
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

```java
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
Triple Pattern Fragments

• Observation:
  – Operating SPARQL endpoints is costly
    • Hence, there are often downtimes
  – Accessing data via dumps or derefencing is time consuming
    • See initial experiment
• Triple Pattern Fragments provide a middle ground solution

http://linkeddatafragments.org
Triple Pattern Fragments

- Only allow simple restrictions
  - i.e., only `{?s ?p ?o}`
- Provide results in a paged fashion
  - Estimated count
  - Links to further pages

Data Portal @ linkeddatafragments.org

DBpedia 2014 Query

subject: http://www.w3.org/1999/02/22-rdf-syntax-ns#type
predicate: http://dbpedia.org/ontology/Restaurant
object: http://dbpedia.org/ontology/Restaurant

Find matching triples

Matches in DBpedia 2014 for `{?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://...}
Showing triples 1 to 100 of 1,213 with 100 triples per page. next

- AsianHound_places type Restaurant.
- t_Brouwerskolkje type Restaurant.
- t_Coenenest type Restaurant.
- t_Koetshuis type Restaurant.
- t_Nisseleharn type Restaurant.
- t_Oosterhuis type Restaurant.
- t_Verhuis type Restaurant.
- 1321_Downtown_Taproom_Bistro type Restaurant.
- 21_Club type Restaurant.
- 36_on_the_Quay type Restaurant.
- 5_North_St type Restaurant.
- 68-86_Bar_and_Restaurant type Restaurant.
- Aan_de_Poel type Restaurant.
- Al_Hoc_(Restaurant) type Restaurant.
- Akelare type Restaurant.
- Alain_Ducasse_at_the_Dorchester type Restaurant.
- Albannach_(restaurant) type Restaurant.

http://linkeddatafragments.org
Triple Pattern Fragments

- Most SPARQL queries can be solved by iteratively retrieving TPFs
  - Successively issuing new selectors
  - More targeted, i.e., less calls, than dereferencing individual URIs

Triple Pattern Fragments

- Example: astronauts born in capital countries

  ```logic
  select ?x ?y ?z where {
    ?x a dbpedia-owl:Astronaut .
    ?x dbpedia-owl:birthPlace ?y .
    ?z a dbpedia-owl:Country
  }
  ```

- Algorithm:
  - retrieve pattern: ?x a dbpedia-owl:Astronaut .
  - for each result ?x: retrieve ?x dbpedia-owl:birthPlace ?y .
  - for each result ?z: check ?z a dbpedia-owl:Country .
Triple Pattern Fragments

• Middle ground between
  – setting up a SPARQL server (costly for the server)
  – providing a full RDF dump (costly for the client)
• In our example, a SPARQL query was broken down into ~3k HTTP GET requests
  – Using clever index structures, this might still be faster
  – Results may also be streamed – allows for early stopping
Triple Pattern Fragments vs. SPARQL

• All SPARQL constructs can be translated to a TPF query plan
• Some are quite fast
  – e.g., typical star-shaped queries
• Some are rather slow
  – e.g., regex queries for labels
Semantic Web – Architecture

here be dragons...

Semantic Web Technologies
(This lecture)

Technical Foundations

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