SPARQL

Querying RDF(S)
Where are we?

- XML
- RDF(S)/SPARQL
- PL/FOL
- OWL
- DL Extensions
- Scalability
- OWL in practice
- Practical Topics
- OWL Reasoning
- OWL in practice
Recap

- Elements of RDF?
- Elements of RDFS?
- What was RDF/XML?
  - Other options?
- What can you call “semantics” in RDF(S)?
- How is the semantics defined?
MOTIVATION
Motivation

Having RDF data available is not enough

Need tools to process, transform, and reason with the information
Need a way to store the RDF data and interact with it

Are existing storage systems appropriate to store RDF data?
Are existing query languages appropriate to query RDF data?
Databases and RDF

Relational database are a well established technology to store information and provide query support (SQL). Relational database have been designed and implemented to store concepts in a predefined (not frequently alterable) schema.

How can we store the following RDF data in a relational database?

```xml
<rdf:Description rdf:about="949318">
  <rdf:type rdf:resource="&uni;lecturer"/>
  <uni:name>Dieter Fensel</uni:name>
  <uni:title>University Professor</uni:title>
</rdf:Description>
```

Several solutions are possible.
**Solution 1: Relational “Traditional” approach**

<table>
<thead>
<tr>
<th>Lecturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>949318</td>
</tr>
</tbody>
</table>

**Approach:** We can create a table “Lecturer” to store information about the “Lecturer” RDF Class.

**Drawbacks:** Every time we need to add new content we have to create a new table - > Not scalable, not dynamic, not based on the RDF principles (TRIPLES)
Databases and RDF

**Solution 2: Relational “Triple” based approach**

**Approach:** We can create a table to maintain all the triples S P O (and distinguish between URI objects and literals objects).

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>ObjectURI</th>
<th>ObjectLiteral</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>102</td>
<td>103</td>
<td>null</td>
</tr>
<tr>
<td>101</td>
<td>104</td>
<td></td>
<td>201</td>
</tr>
<tr>
<td>101</td>
<td>105</td>
<td></td>
<td>202</td>
</tr>
<tr>
<td>103</td>
<td>...</td>
<td>...</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>949318</td>
</tr>
<tr>
<td>102</td>
<td>rdf:type</td>
</tr>
<tr>
<td>103</td>
<td>uni:lecturer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>Dieter Fensel</td>
</tr>
<tr>
<td>202</td>
<td>University Professor</td>
</tr>
<tr>
<td>203</td>
<td>...</td>
</tr>
</tbody>
</table>

**Drawbacks:** We are flexible w.r.t. adding new statements dynamically without any change to the database structure… but what about querying?
State a SQL-query for both approaches, which answers:
Find the names of all the lecturers?
Why Native RDF Repositories?

What happens if I want to find the names of all the lecturers?

**Solution 1:** Relation “traditional” approach:

```sql
SELECT NAME FROM LECTURER
```

We need to query a single table which is easy, quick and performing

No **JOIN** required (the most expensive operation in a db query)

**BUT** we already said that Traditional approach is not appropriate
Why Native RDF Repositories?

What happens if I want to find the names of all the lecturers?

**Solution 2:** Relational “triple” based approach:

```sql
SELECT L.Value FROM Literals AS L
INNER JOIN Statement AS S ON S.ObjectLiteral=L.ID
INNER JOIN Resources AS R ON R.ID=S.Predicate
INNER JOIN Statement AS S1 ON S1.Predicate=S.Predicate
INNER JOIN Resources AS R1 ON R1.ID=S1.Predicate
INNER JOIN Resources AS R2 ON R2.ID=S1.ObjectURI
WHERE R.URI = "uni:name"
AND R1.URI = "rdf:type"
AND R2.URI = "uni:lecturer"
```
Why Native RDF Repositories?

Solution 2
The query is quite complex: 5 JOINs!

This require a lot of optimization specific for RDF and triple data storage, that it is not included in Relational DB

For achieving efficiency a layer on top of a database is required. More, SQL is not appropriate to extract RDF fragments

Do we need a new query language?
Storage for efficient SPARQL queries

1. Giant Triple Store
   - All triples in one table (easy to implements; many self joins)
   - Optionally with IDs, instead of terms

2. Property Tables
   - Properties of similar subjects into one table (fewer joins, NULL values!)

3. Vertically partitioned tables (Binary tables)
   - One table for each property (performance, expensive inserts)

4. Hexastore
   - Create six indexes spo, pos, osp, sop, pso, ops (fast joins, storage!)
Query Languages

*Querying* and inferencing is the very purpose of information representation in a machine-accessible way.

A query language is a language that allows a user to retrieve information from a “data source”

**E.g. data sources**
- A simple text file
- XML file
- A database
- The “Web”

Query languages usually includes *insert* and *update* operations
Example of Query Languages

**SQL**
Query language for relational databases

**XQuery, Xpointer, and XPath**
Query language for XML data sources

**RDQL**
Query language for RDF in Jena models

**SPARQL**
Query language for RDF graphs
Why an RDF Query Language?
Different XML Representations

XML at a lower level of abstraction than RDF
There are various ways of syntactically representing an RDF statement in XML
Thus we would require several XQuery queries, e.g.

//uni:lecturer/uni:title if uni:title element
//uni:lecturer/@uni:title if uni:title attribute
Both XML representations equivalent!
vcards

- File format for electronic business cards
  - Name
  - Address
  - Phone number, email, …

```xml
<?xml version="1.0" encoding="UTF-8"?>
<vcards xmlns="urn:ietf:params:xml:ns:vcard-4.0">
  <vcard>
    <n>
      <surname>Gump</surname>
      <given>Forrest</given>
      <additional/>
      <prefix/>
      <suffix/>
    </n>
    <fn>Forrest Gump</fn>
    <org>Bubba Gump Shrimp Co.</org>
    <title>Shrimp Man</title>
    <photo>http://www.example.com/dir_photos/my_photo.gif</photo>
    <tel>
      <parameters>
        <type>work</type>
        <type>voice</type>
      </parameters>
      <uri>tel:+1-111-555-1212</uri>
    </tel>
  </vcard>
</vcards>
```
Example RDF Graph

```xml
<http://example.org/#john> <http://.../vcard-rdf/3.0#FN> "John Smith"

<http://example.org/#john> <http://.../vcard-rdf/3.0#N> :_X1
_:X1 <http://.../vcard-rdf/3.0#Given> "John"
_:X1 <http://.../vcard-rdf/3.0#Family> "Smith"

<http://example.org/#john> <http://example.org/#hasAge> "32"

<http://example.org/#john> <http://example.org/#marriedTo> <#mary>

<http://example.org/#mary> <http://.../vcard-rdf/3.0#FN> "Mary Smith"

<http://example.org/#mary> <http://.../vcard-rdf/3.0#N> :_X2
_:X2 <http://.../vcard-rdf/3.0#Given> "Mary"
_:X2 <http://.../vcard-rdf/3.0#Family> "Smith"

<http://example.org/#mary> <http://example.org/#hasAge> "29"
```
Task

Transform the RDF/NTriple from the previous slide into a graph-like notation.
Example
Introduction to RDQL

RDF Data Query Language
JDBC/ODBC friendly

Simple:

```
SELECT
  some information
FROM
  somewhere
WHERE
  this match
AND
  these constraints
USING
  these vocabularies
```
Example

q1 contains a query:

```
SELECT ?x
WHERE (?x, <http://www.w3.org/2001/vcard-rdf/3.0#FN>, "John Smith")
```

The outcome is:

```
x
------------------------
<http://somewhere/JohnSmith/>
```
Example

q1 contains a query:

```
SELECT ?x
WHERE (?x, <http://www.w3.org/2001/vcard-rdf/3.0#FN>, "John Smith")
```

Idea:
- Variables are bound to RDF terms
- SELECT basically returns a table (or list)
- WHERE clause: turtle-serialization of triple patterns
Example

Return all the resources that have property FN and the associated values:

```
SELECT ?x, ?fname
WHERE (?x, <http://www.w3.org/2001/vcard-rdf/3.0#FN>, ?fname)
```

The outcome is:

<table>
<thead>
<tr>
<th>x</th>
<th>fname</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://somewhere/JohnSmith/">http://somewhere/JohnSmith/</a></td>
<td>&quot;John Smith&quot;</td>
</tr>
<tr>
<td><a href="http://somewhere/SarahJones/">http://somewhere/SarahJones/</a></td>
<td>&quot;Sarah Jones&quot;</td>
</tr>
<tr>
<td><a href="http://somewhere/MattJones/">http://somewhere/MattJones/</a></td>
<td>&quot;Matt Jones&quot;</td>
</tr>
</tbody>
</table>
Limitations

Does not take into account semantics of RDF(S)

For example:

ex:human rdfs:subClassOf ex:animal
ex:student rdfs:subClassOf ex:human
ex:john rdf:type ex:student

Query: “To which class does the resource John belong?”
Expected answer: ex:student, ex:human, ex:animal, However, the query:

SELECT ?x
WHERE (<http://example.org/#john>, rdf:type, ?x)
USING rdf FOR <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

Yields only:
<http://example.org/#student>

Solution: Inference Engines
A language to query RDF data

SPARQL
SPARQL 1.0

- Allows:
  - Extraction of information from RDF graphs
  - Exploration via queries
  - Complex join operations
  - Transformation of between vocabularies
  - Construction of new RDF graphs
Querying RDF

SPARQL
RDF Query language
Based on RDQL
Uses SQL-like syntax

Example:

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

SELECT ?name
FROM <http://example.org/personal>
WHERE { ?s uni:name ?name.
?s rdf:type uni:lecturer }
SPARQL Query keywords

PREFIX: based on namespaces

DISTINCT: The DISTINCT solution modifier eliminates duplicate solutions. Specifically, each solution that binds the same variables to the same RDF terms as another solution is eliminated from the solution set.

REDUCED: While the DISTINCT modifier ensures that duplicate solutions are eliminated from the solution set, REDUCED simply permits them to be eliminated. The cardinality of any set of variable bindings in an REDUCED solution set is at least one and not more than the cardinality of the solution set with no DISTINCT or REDUCED modifier.

LIMIT: The LIMIT clause puts an upper bound on the number of solutions returned. If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned.
SPARQL Query keywords

OFFSET: OFFSET causes the solutions generated to start after the specified number of solutions. An OFFSET of zero has no effect.

ORDER BY: The ORDER BY clause establishes the order of a solution sequence.

Following the ORDER BY clause is a sequence of order comparators, composed of an expression and an optional order modifier (either ASC() or DESC()). Each ordering comparator is either ascending (indicated by the ASC() modifier or by no modifier) or descending (indicated by the DESC() modifier).
Task

Find an intuitive example for using the REDUCED-clause!
SPARQL Queries: All Full Names

“Return the full names of all people in the graph”

PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?fullName
WHERE {?x vCard:FN ?fullName}

result:

fullName

"John Smith"
"Mary Smith"
SPARQL Queries: Properties

"Return the relation(s) between John and Mary"

PREFIX ex: <http://example.org/#>
SELECT ?p
WHERE {ex:john ?p ex:mary}

result:

p

@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
 vcard:FN "John Smith" ;
 vcard:N [
   vcard:Given "John" ;
   vcard:Family "Smith" ] ;
ex:hasAge 32 ;
ex:marriedTo :mary .
ex:mary
 vcard:FN "Mary Smith" ;
 vcard:N [
   vcard:Given "Mary" ;
   vcard:Family "Smith" ] ;
ex:hasAge 29 .

<http://example.org/#marriedTo>
Task: Purpose of the following query?

PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX ex: <http://example.org/#>

SELECT ?y
WHERE {?x vCard:FN "John Smith".
      ?x ex:marriedTo ?y}
SPARQL Queries: Complex Patterns

“Return the spouse of a person by the name of John Smith”

PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX ex: <http://example.org/#>
SELECT ?y
WHERE {?x vCard:FN "John Smith".
    ?x ex:marriedTo ?y}

result:

<http://example.org/#mary>
SPARQL Queries: Blank Nodes

PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name, ?firstName
WHERE {?x vCard:N ?name .
      ?name vCard:Given ?firstName}

@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [vcard:Given "John" ; vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [vcard:Given "Mary" ; vcard:Family "Smith" ] ;
  ex:hasAge 29 .
SPARQL Queries: Blank Nodes

“Return the first name of all people in the KB”

PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name, ?firstName
WHERE {?x vCard:N ?name .
    ?name vCard:Given ?firstName}

result:

name  firstName
======  =========
_:a  "John"
_:b  "Mary"
SPARQL Queries: Building RDF Graph

“Rewrite the naming information in original graph by using the foaf:name”

PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:name ?name }
WHERE { ?x vCard:FN ?name }

result:

#john foaf:name "John Smith"
#marry foaf:name "Marry Smith"
SPARQL Queries: Building RDF Graph

“Rewrite the naming information in original graph by using the foaf:name”

PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:name ?name }
WHERE { ?x vCard:FN ?name }

result:

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [vcard:Given "John" ; vcard:Family "Smith"] ;
ex:hasAge 32 ;
ex:marriedTo marry.

ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [vcard:Given "Mary" ; vcard:Family "Smith"] ;
  ex:hasAge 29 .
```

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:ex="http://example.org">
  <rdf:Description rdf:about="ex:john">
    <foaf:name>John Smith</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:about="ex:mary">
    <foaf:name>Marry Smith</foaf:name>
  </rdf:Description>
</rdf:RDF>
```
SPARQL Queries: Testing if the Solution Exists

“Are there any married persons in the KB?”

PREFIX ex: <http://example.org/#>
ASK { ?person ex:marriedTo ?spouse }

result:

yes

@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [ vcard:Given "John" ; vcard:Family "Smith" ] ;
ex:hasAge 32 ;
ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [ vcard:Given "Mary" ; vcard:Family "Smith" ] ;
ex:hasAge 29 .

------------------
SPARQL Queries: Constraints (Filters)

“Return all people over 30 in the KB”

PREFIX ex: <http://example.org/#>
SELECT ?x
WHERE {?x hasAge ?age .
FILTER(?age > 30)}

result:

x

<http://example.org/#john>

<http://example.org/#mary>
SPARQL Queries: Constraints (Filters)

- Unary operators for filters in SPARQL:
  - !A
  - BOUND (A)
  - isURI (A)
  - isLITERAL (A)
  - ...

- Can be combined by &&, ||
- Plus regular expressions, equality,
  ...

SPARQL Queries: Optional Patterns

“Return all people and (optionally) their spouse”

PREFIX ex: <http://example.org/#>
SELECT ?person, ?spouse
WHERE {?person ex:hasAge ?age .
OPTIONAL { ?person ex:marriedTo ?spouse } }

result:

?person ?spouse
=============================<http://example.org/#mary><http://example.org/#john> <http://example.org/#mary>

@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [vcard:Given "John" ; vcard:Family "Smith" ] ;
ex:hasAge 32 ;
ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [vcard:Given "Mary" ; vcard:Family "Smith" ] ;
ex:hasAge 29 .
An example of usage of SPARQL

ILLUSTRATION BY A LARGER EXAMPLE
A RDF Graph Modeling Movies

```
movie1
  rdf:type movie:Movie
  movie:title "Edward ScissorHands"
  movie:year "1990"
  movie:hasPart "Edward ScissorHands"
  movie:Role "Edward ScissorHands"
  movie:playedBy r1
  r1 rdf:type movie:Role
    movie:characterName "Edward ScissorHands"

movie:Genre
  rdf:type
  movie:Romance
  rdf:type
  movie:Comedy

```
Example Query 1

Select the movies that have a character called “Edward Scissorhands”

PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x ?t
WHERE {
    ?x movie:title ?t ;
    movie:hasPart ?y .
    ?y movie:characterName ?z .
    FILTER (?z = "Edward Scissorhands"@en)
}
Example Query 1

PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x ?t
WHERE {
    ?x movie:title ?t ;
    movie:hasPart ?y .
    ?y movie:characterName ?z .
    FILTER (?z = "Edward Scissorhands"@en)
}

Note the use of “;” This allows to create triples referring to the previous
triple pattern (extended version would be ?x movie:hasPart ?y)
Note as well the use of the language speciation in the filter @en
Example Query 2

Create a graph of actors and relate them to the movies they play in (through a new ‘playsInMovie’ relation)

PREFIX movie: <http://example.org/movies/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT {
    ?x foaf:firstName ?fname.
    ?x foaf:lastName ?lname.
    ?x movie:playInMovie ?m
}
WHERE {
    ?m movie:title ?t ;
    movie:hasPart ?y .
    ?y movie:playedBy ?x .
    ?x foaf:firstName ?fname.
    ?x foaf:lastName ?lname.
}
Example Query 3

Find all movies which share at least one genre with “Gone with the Wind”

PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x2 ?t2
WHERE {
?x1 movie:title ?t1.
?x1 movie:genre ?g1.
?x2 movie:genre ?g2.
FILTER (?t1 = “Gone with the Wind”@en &&
?x1!=?x2 && ?g1=?g2)
}


Extending SPARQL

SPARQL is still under continuous development (SPARQL 1.1). Current investigate possible extensions include:

Better support for OWL semantics
RDF data insert and update
Aggregate functions
Federated/distributed queries

SELECT (COUNT(?studentID) AS ?C)
WHERE (?studentID attends “Introduction to the Semantic Web“)
SPARQL Protocol

Used to query/respond to SPARQL queries via http

SPARQL URI consists of three parts:
- I. URL of SPARQL endpoint
- II. RDF-Graph(s) to be queries
- III. Query string; query=SELECT …
Why care about scalability?

Rapid growth of available semantic data

> 31 billion triples in the LOD cloud, 325 sources
DBPedia: 3.6 million entities, 1.2 billion triples
Query Optimization

Query Optimization is a process that tends to device a query execution plan that takes the minimum response time.

The response time is minimized by reducing the number of blocks that must be read or be written to (external) memory to complete the query.

Query Optimization is vital especially in cases where numerous transactions are made every second.
SPARQL

SPARQL uses multiple triples to match certain conditions and extract data based on these conditions.

There are two factors that play an important role in the response time of a SPARQL query,
- The order in which the triples are accessed,
- The necessity of each triple

SPARQL Optimization also depends on the platform on which it is implemented.
Types of Optimization

There are two types of Query Optimization,

- Logical Optimization
- Physical Optimization

Logical Optimization generates a sequence or an order in which the triples are processed so as to minimize the response time.

Physical Optimization is a high level optimization where we determine how each operation is done.
Selectivity

Selectivity of a triple pattern is the fraction of triples matching the pattern. This helps making a decision on the execution plan.

Consider the following query,

```
?x  NS:type    NS:animal
?x  NS:species "zebra"
```

Changing the order in which they are executed can save us a lot of time.
Summary Statistics

It is a traditional practice to keep track of metadata, i.e. data about the data in order to calculate cardinalities, generate indices of data etc.

These metadata can be used to create summary statistics in such a way to facilitate the estimation of the size or result set of any query.

It can also help in calculating selectivity for a particular component of a triple pattern.

Histograms can be used to represent the distribution of data.
Limitations – Logical Optimization

Scalability is the basic limitation of query optimization based on selectivity, as it is not feasible to find selectivity in a dataset containing millions of triples.

The major limitation of logical optimization is the use of special modifiers like, OPTIONAL, UNION, FILTER etc . . . These modifiers affect the selectivity and undermines any algorithm for logical optimization.

Another important issue to take into account is the type of ontology used and the pattern in which data is stored. These cannot be generalized as it varies with each dataset.
Physical Optimization

Physical Optimization is a customized solution for a specific ontology or framework. Here it is decided how each and every query can be implemented based on the ontology and the data.
SUMMARY
SPARQL Summary

RDF Repositories
- Optimized solutions for storing RDF
- Adopts different implementation techniques
- Choice has to be made by multiple factors
  - In general, there is no single solution better than all other ones

SPARQL
- Query language for RDF represented data
- More powerful than XPath based languages
- Supported by a communication protocol
SPARQL Summary

- Expressive graph query language

Pro
- Enables inferencing, Many implementations
- Easy to use; similarity to SQL
- RDF/RDF(S) -> SPARQL

Con
- No notion of privacy (not a real problem!)
- Scalability not always clear
- Locality as a problem: what is relevant for reasoning?