Lecture 4

RDF Stores: SPARQL

Martin Svoboda
svoboda@ksi.mff.cuni.cz

24. 10. 2017

Charles University in Prague, Faculty of Mathematics and Physics
Czech Technical University in Prague, Faculty of Electrical Engineering
Lecture Outline

RDF stores

- Introduction
- Linked Data

SPARQL query language

- Graph patterns
- Filter constraints
- Solution modifiers
- Aggregation
- Query forms
RDF Stores

Data model

• **RDF triples**
  - Components: *subject*, *predicate*, and *object*
  - Each triple represents a *statement* about a real-world entity

• Triples can be viewed as **graphs**
  - *Vertices* for subjects and objects
  - *Edges* directly correspond to individual statements

Query language

• **SPARQL**: *SPARQL Protocol and RDF Query Language*

Representatives

• Apache *Jena*, *rdf4j* (Sesame), Algebraix
• *Multi-model*: *MarkLogic*, OpenLink *Virtuoso*
Linked Data

Linked Data

• Method of publishing structured and interlinked data in a way that allows for an automated processing by programs rather than browsing by human readers

Principles of Linked Open Data

• Identify resources using URIs or even better using URLs
• Publish data about resources in standard formats via HTTP
• Mutually interlink resources to form Web of Data
• Release the data under an open licence
Linked Open Data Cloud

May 2007

Source: http://lod-cloud.net/
Linked Open Data Cloud

September 2011

Source: http://lod-cloud.net/
Linked Open Data Cloud

August 2017

Source: http://lod-cloud.net/
Linked Data

Statistics

- October 2007
  - 25 datasets
  - 2 billion triples, 2 million links
- September 2011
  - 295 datasets
  - 31 billion triples, 504 million links
- August 2017
  - 1163 datasets
SPARQL Query Language
SPARQL Query Language

- Query language for RDF data
  - **Graph patterns**, optional graph patterns, **subqueries**, negation, **aggregation**, value constructors, ...
- Versions: 1.0 (2008), **1.1** (2013)
- W3C recommendations
  - [https://www.w3.org/TR/sparql11-query/](https://www.w3.org/TR/sparql11-query/)
  - Altogether 11 recommendations: query language, update facility, federated queries, protocol, result formats, ...
Sample Data

Graph of movies <http://db.cz/movies>

@prefix i: <http://db.cz/terms#> .
@prefix m: <http://db.cz/movies/> .
@prefix a: <http://db.cz/actors/> .

m:vratnelahve
  rdf:type i:Movie ; i:title "Vratné lahve" ;
  i:year "2006" ;
  i:actor a:sverak , a:machacek .

m: samotari
  rdf:type i:Movie ; i:title "Samotáři" ;
  i:year "2000" ;
  i:actor a:schneiderova , a:trojan , a:machacek .

m: medvidek
  rdf:type i:Movie ; i:title "Medvídek" ;
  i:year "2007" ;
  i:actor a:machacek , a:trojan ;
  i:director "Jan Hřebejk" .

m: zelary
  rdf:type i:Movie .
Sample Data

Graph of actors <http://db.cz/actors>

@prefix i: <http://db.cz/terms#> .
@prefix a: <http://db.cz/actors/> .
a:trojan
   rdf:type i:Actor ;
   i:firstname "Ivan" ; i:lastname "Trojan" ;
   i:year "1964" .
a:machacek
   rdf:type i:Actor ;
   i:firstname "Jiří" ; i:lastname "Macháček" ;
   i:year "1966" .
a:schneiderova
   rdf:type i:Actor ;
   i:firstname "Jitka" ; i:lastname "Schneiderová" ;
   i:year "1973" .
a:sverak
   rdf:type i:Actor ;
   i:firstname "Zdeněk" ; i:lastname "Svěrák" ;
   i:year "1936" .
Sample Query

Find all movies, return their titles and years they were filmed

```
PREFIX i: <http://db.cz/terms#>
SELECT ?t ?y
FROM <http://db.cz/movies>
WHERE

  { 
    ?m rdf:type i:Movie ;
    i:title ?t ;
    i:year ?y .
  }
ORDER BY ?y
```

<table>
<thead>
<tr>
<th>?t</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samotáři</td>
<td>2000</td>
</tr>
<tr>
<td>Vratné lahve</td>
<td>2006</td>
</tr>
<tr>
<td>Medvídek</td>
<td>2007</td>
</tr>
</tbody>
</table>
Sample Query
Graph Pattern Matching

Graph patterns

- **Basic graph pattern**
  - Based on ordinary **triples with variables**
    - ?variable or $variable
- More complicated graph patterns
  - E.g. group, optional, minus, ...

Graph pattern matching

- Our goal is to find all **subgraphs of the data graph that are matched by the query graph pattern**
  - I.e. subgraphs of the data graph that are identical to the query graph pattern with variables substituted by particular terms
- One matching subgraph = one **solution** = one row of a table
**Graph Pattern Matching**

**Query result** = **solution sequence** = ordered multiset of solutions

<table>
<thead>
<tr>
<th>?t</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samotáři</td>
<td>2000</td>
</tr>
<tr>
<td>Vratné lahve</td>
<td>2006</td>
</tr>
<tr>
<td>Medvídek</td>
<td>2007</td>
</tr>
</tbody>
</table>

```{ 
  { (?t, "Samotáři"), (?y, "2000") },
  { (?t, "Vratné lahve"), (?y, "2006") },
  { (?t, "Medvídek"), (?y, "2007") }
} 
```

**Solution** = set of variable bindings

<table>
<thead>
<tr>
<th>?t</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samotáři</td>
<td>2000</td>
</tr>
</tbody>
</table>

```{ 
  { (?t, "Samotáři"), (?y, "2000") }
} 
```

**Variable binding** = pair of a variable name and a value it is assigned

<table>
<thead>
<tr>
<th>?t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samotáři</td>
</tr>
</tbody>
</table>
Graph Pattern Matching

Compatibility of solutions

- Two solutions are mutually compatible if and only if all the variables they share are pairwise bound to identical values.

Examples

- Compatible solutions
  - \{ (\texttt{?m}, \texttt{m:samotari}), (\texttt{?t}, "Samotáři") \}
  - \{ (\texttt{?m}, \texttt{m:samotari}), (\texttt{?y}, "2000") \}

- Incompatible solutions
  - \{ (\texttt{?m}, \texttt{m:samotari}), (\texttt{?t}, "Samotáři") \}
  - \{ (\texttt{?m}, \texttt{m:medvidek}), (\texttt{?y}, "2007") \}
**Select Queries**

**SELECT** queries

- **Prologue declarations** – `PREFIX`, `BASE`
- **Main clauses**
  - `SELECT` – variables to be projected
  - `FROM` – data graphs to be queried
  - `WHERE` – graph patterns to be matched
- **Solution modifiers** – `ORDER BY`, ...
Prologue Declarations

Prologue declarations

• Allow to simplify IRI references by declaring base IRIs

BASE clause

• One single base IRI is defined
  all relative IRI references are then related to this base IRI

PREFIX clause

• Several base IRIs are defined, each is associated with a name
  all prefixed names are then related to the respective base IRI
Prologue Declarations

Examples

• When BASE <http://db.cz/> is defined, then a relative IRI reference terms#Movie is interpreted as http://db.cz/terms#Movie

• When PREFIX i: <http://db.cz/> is defined, then a prefixed name i:terms#Movie is interpreted as http://db.cz/terms#Movie
Where Clause

WHERE clause

- Prescribes one **group graph pattern**

Types of **graph patterns**

- **Basic** – triple patterns to be matched
- **Group** – set of graph patterns to be matched
- **Optional** – graph pattern to be matched only if possible
- **Alternative** – two or more alternative graph patterns
- **...**

Graph patterns can be inductively combined into complex ones
Graph Patterns

Basic Graph Pattern

**Basic** graph pattern (triple block)

- One or more triple patterns to be all matched

- Ordinary **triples** separated by .
- ... or their abbreviated forms inspired by Turtle notation
  - **Object lists** using ,
  - **Predicate-object lists** using ;
  - **Blank nodes** using []

Examples

- s p1 o1 . s p1 o2 . s p2 o3 .
- s p1 o1 , o2 ; p2 o3 .
Graph Patterns

Basic Graph Pattern

Interpretation

• **All the involved triple patterns must be matched**
  - I.e. we combine them as if they were in conjunction
  - More precisely...
    - Each triple pattern is evaluated to its solution sequence
    - All combinations of compatible solutions are then found

• **Note that all the variables need to be bound**
  - I.e. if any of the involved variables cannot be bound at all, then the entire basic graph pattern cannot be matched!
Graph Patterns: Example

Basic Graph Pattern

Titles and years of all movies

```sparql
PREFIX i: <http://db.cz/terms#>
SELECT ?t ?y
FROM <http://db.cz/movies>
WHERE {
  ?m rdf:type i:Movie . # triple 1
  ?m i:title ?t .      # triple 2
  ?m i:year ?y .      # triple 3
}
```

<table>
<thead>
<tr>
<th>?t</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vratné lahve</td>
<td>2006</td>
</tr>
<tr>
<td>Samotáři</td>
<td>2000</td>
</tr>
<tr>
<td>Medvídek</td>
<td>2007</td>
</tr>
</tbody>
</table>
Graph Patterns: Example

Basic Graph Pattern

\[
\begin{align*}
[t_1] &= 
\begin{array}{l}
?m \\
m:vratnelahve \\
m:samotari \\
m:medvidek \\
m:zelary
\end{array} \\

[t_2] &= 
\begin{array}{|c|c|}
\hline
?m & ?t \\
\hline
m:vratnelahve & Vratné lahve \\
m:samotari & Samotáři \\
m:medvidek & Medvídek \\
\hline
\end{array}
\]

[t_3] &= 
\begin{array}{|c|c|}
\hline
?m & ?y \\
\hline
m:vratnelahve & 2006 \\
m:samotari & 2000 \\
m:medvidek & 2007
\hline
\end{array}
\]

<table>
<thead>
<tr>
<th>?t</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vratné lahve</td>
<td>2006</td>
</tr>
<tr>
<td>Samotáři</td>
<td>2000</td>
</tr>
<tr>
<td>Medvídek</td>
<td>2007</td>
</tr>
</tbody>
</table>
Equivalence of literals

- Values must be identical
- And when `types / language tags` are specified, these must be identical as well
  - E.g.: "Medvídek"@cs ≠ "Medvídek"

Equivalence of blank nodes

- Blank nodes in query patterns act as `non-selectable variables`
- **Labels of blank nodes** in data graphs / query graph patterns / query results may not refer to the same nodes despite being the same
  - I.e. the scope of validity is always local only
Graph Patterns

Group graph pattern

Set of graph patterns to be all matched

- nested SELECT query
- triples
- group graph pattern
- OPTIONAL graph pattern
- UNION graph pattern
- MINUS graph pattern
- GRAPH graph pattern
- BIND
- FILTER
- triples
Graph Patterns

Group Graph Pattern

Two modes

- **Nested SELECT query**
  - Only with SELECT and WHERE clauses and solution modifiers
    i.e. without FROM clause

- **Set of graph patterns** interleaved by **triple blocks**

Interpretation

- **All the involved graph patterns must be matched**
  - I.e. we combine them as if they were in conjunction

Notes

- Empty group patterns `{}` are also allowed
Graph Patterns

Optional Graph Pattern

**OPTIONAL** graph pattern

One group graph pattern is tried to be matched

Interpretation

- When the optional part does not match, it creates no bindings but does not eliminate the solution
Graph Patterns: Example

Optional Graph Pattern

Movies together with their directors when possible

```
PREFIX i: <http://db.cz/terms#>
SELECT ?t ?y ?d
FROM <http://db.cz/movies>
WHERE
  {
    ?m rdf:type i:Movie ;
      i:title ?t ;
      i:year ?y .
    OPTIONAL { ?m i:director ?d . }
  }
```

<table>
<thead>
<tr>
<th>?t</th>
<th>?y</th>
<th>?d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vratné lahve</td>
<td>2006</td>
<td></td>
</tr>
<tr>
<td>Samotáři</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>Medvídek</td>
<td>2007</td>
<td>Jan Hřebejk</td>
</tr>
</tbody>
</table>
**Graph Patterns**

**Alternative Graph Pattern**

**UNION** graph pattern

Two or more group graph patterns are tried to be matched

**Interpretation**

- Standard set **union** of the involved query results
Graph Patterns

Minus Graph Pattern

**MINUS** graph pattern

One group graph pattern removing compatible solutions

Interpretation

- Solutions of the first pattern are preserved if and only if they are not compatible with any solution of the second pattern
  - I.e. minus graph pattern does not correspond to the standard set minus operation!
Graph Patterns: Example

Minus Graph Pattern

Titles of movies that have no director

PREFIX i: <http://db.cz/terms#>
SELECT ?t
FROM <http://db.cz/movies>
WHERE
{
  ?m rdf:type i:Movie ;
  i:title ?t .
  MINUS { ?m rdf:type i:Movie ; i:director ?d . } # pattern 2
}

?t
Vratné lahve
Samotáři
**Graph Patterns: Example**

**Minus Graph Pattern**

\[
[p_1] = \begin{array}{|c|c|}
\hline
?m & ?t \\
\hline
m:vratnelahve & Vratné lahve \\
m:samotari & Samotáři \\
m:medvidek & Medvídek \\
\hline
\end{array}
\]

\[
[p_2] = \begin{array}{|c|c|}
\hline
?m & ?d \\
\hline
m:medvidek & Jan Hřebejk \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
?t \\
\hline
\end{array}
\]

- Vratné lahve
- Samotáři

---

From Clause

**FROM clause**

- Defines data graphs to be queried

**Dataset** = collection of graphs to be queried

- One default graph
  - Merge of all the declared graphs from unnamed FROM clauses
  - Empty when no unnamed FROM clause is provided

- Zero or more named graphs

**Active graph** = used for the evaluation of graph patterns

- The default graph unless changed using GRAPH graph pattern
From Clause: Example

Names of actors who played in Medvídek movie

```
PREFIX i: <http://db.cz/terms#>
PREFIX m: <http://db.cz/movies/>
SELECT ?f ?l
FROM <http://db.cz/movies>
FROM <http://db.cz/actors>
WHERE
{
  m:medvidek i:actor ?a .
}
```

<table>
<thead>
<tr>
<th>?f</th>
<th>?l</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jiří</td>
<td>Macháček</td>
</tr>
<tr>
<td>Ivan</td>
<td>Trojan</td>
</tr>
</tbody>
</table>
Graph Patterns

Graph Graph Pattern

**GRAPH** graph pattern

Pattern evaluated with respect to a particular named graph

- **Changes the active graph** for a given group graph pattern
  - `GRAPH <http://db.cz/actors> { ... }`
- We can also consider all the named graphs
  - `GRAPH ?g { ... }`
Graph Patterns: Example

Graph Graph Pattern

Names of actors who played in *Medvídek* movie

```sparql
PREFIX i: <http://db.cz/terms#>
PREFIX m: <http://db.cz/movies/>
SELECT ?f ?l
FROM <http://db.cz/movies>
FROM NAMED <http://db.cz/actors>
WHERE {
  m:medvidek i:actor ?a .
  GRAPH <http://db.cz/actors> {
  }
}
```

<table>
<thead>
<tr>
<th>?f</th>
<th>?l</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jiří</td>
<td>Macháček</td>
</tr>
<tr>
<td>Ivan</td>
<td>Trojan</td>
</tr>
</tbody>
</table>
Variable Assignments

**BIND graph pattern**

Explicitly assigns a value to a given variable

- This variable **must not** yet be bound!
Filter Constraints

**FILTER constraints**

Impose constraints on variables and their values

- Only solutions satisfying the given condition are preserved
- Does not create any new variable bindings!
- Always applied on the entire group graph pattern i.e. evaluated at the very end
Filter Constraints: Example

Movies filmed in 2005 or later where Ivan Trojan played

```
PREFIX i: <http://db.cz/terms#>
PREFIX a: <http://db.cz/actors/>
SELECT ?t ?y
FROM <http://db.cz/movies>
WHERE
{
  ?m rdf:type i:Movie ;
  i:title ?t ;
  i:year ?y .
  FILTER (?y >= 2005) &&
  EXISTS { ?m i:actor a:trojan . }
}
```

<table>
<thead>
<tr>
<th>?t</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medvídek</td>
<td>2007</td>
</tr>
</tbody>
</table>
Filter Constraints

Relational expressions

- **Comparisons**
  - $=$, $!=$, $<$, $\leq$, $\geq$, $>$
  - Unbound variable < blank node < IRI < literal

- **Set membership tests**
  - IN and NOT IN

Numeric expressions

- Unary / binary **arithmetic operators** $+$, $-$, $\ast$, $/$
Filter Constraints

Primary expressions

• **Literals** – numeric, boolean, RDF triples
• **Variables**
• **Built-in calls**
• **Parentheses**

Boolean expressions

• Logical connectives
  ▪ Conjunction `&&`, disjunction `||`, negation `!`
• **3-value logic** because of unbound variables (NULL values)
  ▪ `true`, `false`, `error`
Filter Constraints

Built-in calls

• **Term accessors**
  - \texttt{STR} – lexical form of an IRI or literal
  - \texttt{LANG} – language tag of a literal
  - \texttt{DATATYPE} – data type of a literal

• **Variable tests**
  - \texttt{BOUND} – true when a variable is bound to a value
  - \texttt{isIRI}, \texttt{isBLANK}, \texttt{isLITERAL}
Filter Constraints

Built-in calls

- **Existence tests**
  - EXISTS
    - True when a provided group graph pattern is evaluated to at least one solution
  - NOT EXISTS

![Diagram of filter constraints with nodes for EXISTS and NOT, and an arrow connecting them to a group graph pattern.]
**Select clause**

**SELECT clause**

- Enumerates variables to be included in the query result

- Asterisk `*` selects all the variables

**Solution modifiers**

- **DISTINCT** – *duplicate solutions are removed*
- **REDUCED** – some duplicate solutions may be removed (implementation-dependent behavior)
Solution Modifiers

Solution modifiers – modify the entire solution sequence

- Aggregation
  - GROUP BY and HAVING
- Ordering
  - ORDER BY
  - LIMIT and OFFSET
Solution Modifiers

**ORDER BY** clause

- Defines the order of solutions within the query result

- \( \text{ASC}(\ldots) = \text{ascending} \) (default)
- \( \text{DESC}(\ldots) = \text{descending} \)
Solution Modifiers

**LIMIT** clause

- **Limits the number of solutions** in the query result

**OFFSET** clause

- **Skips a certain number of solutions** in the query result
Solution Modifiers: Example

```
PREFIX i: <http://db.cz/terms#>
SELECT ?t ?y
FROM <http://db.cz/movies>
WHERE
{
    ?m rdf:type i:Movie ;
        i:title ?t ;
        i:year ?y .
}
ORDER BY DESC(?y) ASC(?t)
OFFSET 1
LIMIT 5
```

<table>
<thead>
<tr>
<th>?t</th>
<th>?y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vratné lahve</td>
<td>2006</td>
</tr>
<tr>
<td>Samotáři</td>
<td>2000</td>
</tr>
</tbody>
</table>
Aggregation

**GROUP BY + HAVING clauses**

- Standard aggregation over a solution sequence
Aggregation: Example

Numbers of actors in movies with at most 2 actors

PREFIX i: <http://db.cz/terms#>
SELECT ?t (COUNT(?a) AS ?c)
FROM <http://db.cz/movies>
WHERE
{
    ?m rdf:type i:Movie ;
    i:title ?t ;
    i:actor ?a .
}
GROUP BY ?m ?t
HAVING (?c <= 2)
ORDER BY ?c ?t

<table>
<thead>
<tr>
<th>?t</th>
<th>?c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medvídek</td>
<td>2</td>
</tr>
<tr>
<td>Vratné lahve</td>
<td>2</td>
</tr>
</tbody>
</table>
Aggregate functions

- **COUNT**
  - DISTINCT
  - expression

- **SUM**
  - DISTINCT
  - expression

- **MIN**
  - DISTINCT
  - expression

- **MAX**
  - DISTINCT
  - expression

- **AVG**
  - DISTINCT
  - expression

- **GROUP_CONCAT**
  - DISTINCT
  - expression

  - SEPARATOR
  - =
  - string
Query Forms

Query forms

- **SELECT**
  - Finds solutions matching a provided graph pattern
- **ASK**
  - Checks whether at least one solution exists
- **DESCRIBE**
  - Retrieves a graph with data about selected resources
- **CONSTRUCT**
  - Creates a new graph according to a provided pattern
SELECT query form

Finds solutions matching a provided graph pattern

Result

- **Solution sequence** = ordered multiset of solutions
CONSTRUCT query form

Creates a new graph according to a provided pattern

Result

- **RDF graph** constructed according to a group graph pattern
  - Unbound or invalid triples are not involved
**Query Forms: Example**

**CONSTRUCT**

```
PREFIX i: <http://db.cz/terms#>

CONSTRUCT
{
  ?a i:name concat(?f, " ", ?l) .
}

FROM <http://db.cz/actors>

WHERE
{
  ?a rdf:type i:Actor ;
  i:firstname ?f ;
  i:lastname ?l .
}
```

```
<http://db.cz/actors/trojan> i:name "Ivan Trojan" .
```
Lecture Conclusion

SPARQL

- **Query forms**
  - SELECT, ASK, DESCRIBE, CONSTRUCT

- **Graph patterns**
  - Basic, group, optional, alternative, minus
  - Variable assignments
  - Filters

- **Solution modifiers**
  - DISTINCT, REDUCED
  - GROUP BY, HAVING
  - ORDER BY, LIMIT, OFFSET