Query Languages for the Semantic Web?

How to access information that was specified in RDF or OWL?

- Querying information in RDF(S): Simple/RDF/RDFS entailment
  - “Can a certain RDF graph be derived from the given data?”

- Querying information in OWL: Logical entailment
  - “Can a subclass relation be derived from the ontology?”
  - “What are the instances of a given OWL class?”
Are OWL and RDF enough?

Even OWL (2) is too weak for many queries:

- “Who lives together with their parents?” (logical expressivity)
- “Who has married parents?” (logical expressivity)
- “Which properties connect two given individuals?” (schema-level query)
- “Which strings in the ontology are in French language?” (datatype expressivity)
Requirements for Query Languages

- Large query expressivity
- Well-specified semantics

But also:
- Tool support (trade-off with large expressivity)
- Result restriction and manipulation
- Result formatting options
- Machine-readable syntax for queries and results
SPARQL [sparkle]:

**SPARQL Protocol And RDF Query Language**

- W3C specification since Jan 15 2008
- Query language for data from RDF documents
- Extremely successful in practice
- Update in progress (new SPARQL WG since Feb'09)

**Parts of the SPARQL specification:**

- Query language: discussed here
- Result format: encode results in XML
- Query protocol: transmitting queries and results
Basic Queries

A simple example query:

```
PREFIX ex: <http://example.org/>
SELECT ?title ?author
WHERE
  ?book ex:author ?author . }
```

- Main part is a **query pattern** (**WHERE**)
  - Patterns use RDF Turtle syntax
  - Variables can be used, even in predicate positions (**?variable**)
- **Abbreviations** for URIs (**PREFIX**)
- Query result based on **selected variables** (**SELECT**)
A simple example document:

```
@prefix ex: <http://example.org/> .
ex:SemanticWeb
  ex:publishedBy <http://crc-press.com/uri> ;
ex:title "Foundations of Semantic Web Technologies" ;
```

Query results are **tables**, each row is one query result:

<table>
<thead>
<tr>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Foundations of …”</td>
<td><a href="http://example.org/Hitzler">http://example.org/Hitzler</a></td>
</tr>
<tr>
<td>“Foundations of …”</td>
<td><a href="http://example.org/Kr%C3%B6tzsch">http://example.org/Krötzsch</a></td>
</tr>
<tr>
<td>“Foundations of …”</td>
<td><a href="http://example.org/Rudolph">http://example.org/Rudolph</a></td>
</tr>
</tbody>
</table>
Blank Nodes in SPARQL

What do bnodes in SPARQL mean?

**Blank nodes in query patterns:**
- Allowed as subject or object
- Behave like variables that cannot be selected
- bnode ID irrelevant – should not occur more than once per query

**Blank nodes in query results:**
- Placeholder for unknown elements
- ID arbitrary, but may indicate relations between results:

<table>
<thead>
<tr>
<th>subject</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:a</td>
<td>“for”</td>
</tr>
<tr>
<td>_:b</td>
<td>“example”</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>subject</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:y</td>
<td>“for”</td>
</tr>
<tr>
<td>_:g</td>
<td>“example”</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>subject</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:z</td>
<td>“for”</td>
</tr>
<tr>
<td>_:z</td>
<td>“example”</td>
</tr>
</tbody>
</table>
Simple graph patterns are grouped with \{  \} 

Example:

\[
\{ \{ \ ?book \text{ ex:publishedBy} \ <http://crc-press.com/uri> \ . \\
\ ?book \text{ ex:title} \ ?title \ . \} \\
\} \\
\ ?book \text{ ex:author} \ ?author
\]

→ Useful with additional query features
Optional parts can be specified with `OPTIONAL`.

Example:

OPTIONAL { ?book ex:author ?author . }
}
```

→ Parts of the result can be **unbound**:  

<table>
<thead>
<tr>
<th>book</th>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>http://example.org/book1</code></td>
<td>“title 1”</td>
<td><code>http://example.org/johndoe</code></td>
</tr>
<tr>
<td><code>http://example.org/book2</code></td>
<td>“title 2”</td>
<td></td>
</tr>
<tr>
<td><code>http://example.org/book3</code></td>
<td>“title 3”</td>
<td><code>_a</code></td>
</tr>
<tr>
<td><code>http://example.org/book4</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alternative Patterns

Alternatives can be specified with \texttt{UNION}

Example:

\begin{verbatim}
  { ?book ex:author ?author . }
}
\end{verbatim}

\textarrow Result = union of results for one of the alternatives
\textarrow Parts of the result can be \texttt{unbound}

Note: no interaction between multiple variable occurrences in alternative query parts
What does the following mean?

\[
\{ \text{?book ex:publishedBy <http://crc-press.com/uri>} . \\
\{ \text{?book ex:author ?author .} \} \text{ UNION} \\
\{ \text{?book ex:writer ?author .} \} \text{ OPTIONAL} \\
\{ \text{?autor ex:surname ?name .} \}
\}
\]

- Union of two patterns, with an optional condition \textbf{or}
- Union of two patterns, the second of which includes an optional?

→ First interpretation is correct:

\[
\{ \text{?book ex:publishedBy <http://crc-press.com/uri>} . \\
\{ \{ \text{?book ex:author ?author .} \} \text{ UNION} \\
\{ \text{?book ex:writer ?author .} \} \text{ OPTIONAL} \{ \text{?autor ex:surname ?name .} \}
\}
\]
Combining OPTIONAL and UNION

General rules:

- **OPTIONAL** refers to exactly one grouped pattern to the right
- **OPTIONAL** and **UNION** refer to all expressions to their left (left associative), and neither has precedence over the other

Example:

```plaintext
{ {s1 p1 o1} OPTIONAL {s2 p2 o2} UNION {s3 p3 o3}
  OPTIONAL {s4 p4 o4} OPTIONAL {s5 p5 o5}
}
```

means

```plaintext
{ { { {s1 p1 o1} OPTIONAL {s2 p2 o2}
   } UNION {s3 p3 o3}
   } OPTIONAL {s4 p4 o4}
   } OPTIONAL {s5 p5 o5}
}
```
Filters

Additional “filter conditions” can be specified with **FILTER**

Example:

```
  FILTER( (?price < 17) && !isBlank(?book) )
}
```

→ Filter condition: “price a number below 17 and book not a blank node”
→ Results that do not match the filter are removed

**SPARQL** provides many filter functions:
Comparisons (\(=\), \(<\), \(>\), \(<=\), \(>=\), \(!=\)), arithmetics (+, -, *, /), Booleans (&&, ||, !), RDF-specific functions (isLiteral(), Lang(), BOUND(), ...)

Markus Krötzsch, Sebastian Rudolph: Semantic Web Modeling Languages, ESSLLI 2009, Bordeaux
SPARQL: Summary / More Features

- Based on matching simple graph patterns
- Grouping, optionals, and alternatives
- Filters: “extra-logical” result restrictions

Further features:

- Modifiers: postprocess query result set
  
  E.g.: ORDER BY ?age LIMIT 10 OFFSET 5
  
  (→ order by ?age and return 10 results, starting at result 5)

- Result formats: choose encoding of results
  
  E.g.: SELECT ?name, ?age  (→ as in earlier examples)
  
  CONSTRUCT {?name ex:hasAge ?age .}
  
  (→ construct RDF graph as result)
Informal meaning of queries leaves open questions:

- User: “Which results can I expect?”
- Developer: “How should my software behave?”
- Vendor: “Is my product SPARQL-conformant?”

...and, most importantly:

- Computer Scientist: “What's the worst-case complexity of SPARQL?”
Semantics for formal logic:

- Model theoretic semantics: Which interpretations satisfy a knowledge base?
- Proof theoretic semantics: What are valid derivations from a knowledge base?
- ...

Semantic of programming languages:

- Axiomatic Semantics: Which logical statements are valid for a given program?
- Operational Semantics: What effects does the execution of a program have?
- Denotational semantics: How can a program be described as an abstract function?
- ...

What to do with query languages?
Semantics of Query Languages (2)

- Query entailment
  - Queries as *descriptions* of valid results
  - Database as set of logical statements (*theory*)
  - Result as logical *conclusion*

Examples: OWL and RDF as query languages, deductive databases (datalog)

- Query algebra
  - Query as a *procedure* for calculating results
  - Database as *input*
  - Results as *output*

Examples: Relational algebra for SQL, **SPARQL** algebra
Mapping to SPARQL Algebra

  FILTER (?price < 15)
  OPTIONAL
  { ?book ex:author ex:Shakespeare . } UNION
  { ?book ex:author ex:Marlowe . }
}

Semantics of a SPARQL query:
1) Translate query to algebraic expression
2) Evaluate algebraic expression
Mapping to SPARQL Algebra: \textit{BGP}

First step: \textbf{Replacing simple graph patterns}

- Operator \textit{BGP}

- Also resolve URI abbreviations

```sparql
  FILTER (?price < 15) 
  OPTIONAL 
    { BGP(?book <http://eg.org/title> ?title .) } 
} 
UNION 
{ BGP(?book <http://eg.org/author> <http://eg.org/Marlowe> .) } 
```
Second step: **Merging alternative graph patterns**

- **Operator** `Union`
- Refering to patterns next to `UNION` (binding stronger than conjunction!)
- Grouping of multiple alternatives as discussed earlier

```sparql
  FILTER (?price < 15)
  OPTIONAL
  { BGP(?book <http://eg.org/title> ?title .) }
  Union({ BGP(?book <http://eg.org/author>
           <http://eg.org/Shakespeare> .) },
        { BGP(?book <http://eg.org/author>
           <http://eg.org/Marlowe> .) })
}
```
Mapping to SPARQL Algebra

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Join}(P_1, P_2) )</td>
<td>results ( P_1 ) and ( P_2 ) are combined conjunctively</td>
</tr>
<tr>
<td>( \text{Filter}(F, P) )</td>
<td>filter expression ( F ) is applied to result ( P )</td>
</tr>
<tr>
<td>( \text{LeftJoin}(P_1, P_2, F) )</td>
<td>results of ( P_1 ) are conjunctively combined with the results of ( P_2 ) and the filter condition ( F ) is applied; results of ( P_1 ) that are eliminated by this operation are directly added to the output</td>
</tr>
<tr>
<td>( Z )</td>
<td>Constant for the empty expression</td>
</tr>
</tbody>
</table>

Remaining translation stepwise, from inside to outside:

1. Select an innermost graph pattern \( P \)
2. Remove all filter conditions from \( P \)
   \( GF := \) conjunction of all filter conditions
3. Initialise \( G := Z \) and process all subexpressions \( SE \) of \( P \):
   - If \( SE = \text{OPTIONAL} \text{Filter}(F, A) \) : \( G := \text{LeftJoin}(G, A, F) \)
   - Otherwise, if \( SE = \text{OPTIONAL} A \) : \( G := \text{LeftJoin}(G, A, \text{true}) \)
   - Otherwise: \( G := \text{Join}(G, SE) \)
4. If \( GF \) is not empty: \( G := \text{Filter}(GF, G) \)
Mapping to SPARQL Algebra: Joins

  FILTER (?price < 15) 
  OPTIONAL 
    { BGP(?book <http://eg.org/title> ?title .) } 
  Union({ BGP(?book <http://eg.org/author> 
            <http://eg.org/Shakespeare> .) }, 
    { BGP(?book <http://eg.org/author> 
            <http://eg.org/Marlowe> .) }) 
}
Mapping to SPARQL Algebra: Joins

```sparql
{  
  FILTER (?price < 15)  
  OPTIONAL  
  Union(  
    Join(Z,BGP(?book <http://eg.org/author>  
                 <http://eg.org/Shakespeare>  .) ),  
    Join(Z,BGP(?book <http://eg.org/author>  
                 <http://eg.org/Marlowe>  .) )  
  ) 
}
```
Filter((?price < 15),
    Join(
      LeftJoin(
        true
      ),
      )
    )
  )
)
How are SPARQL algebra operators defined?

**Output:**
- “Result table”

**Input:**
- Queried RDF database
- Partial results from subexpressions
- Various parameters for certain operations

How should “results” be represented formally?
Intuition: results encode tables with variable assignments

- **Result:**
  List of solutions (solution sequence)
  → each solution corresponds to one table row

- **Solution:**
  Partial mapping (function)
  - Domain: set of selected variables
  - Range: URIs ∪ blank nodes ∪ RDF literals
  → unbound variables are those that are not assigned values in a solution
The Empty Expression $Z$

Which solution is represented by the empty expression $Z$?

- Domain: $\emptyset$ (no results selected)
- Solutions: exactly one (there is one and only one function with empty domain)

→ “Table with one row but no columns”
Calculating Basic Graph Patterns

A partial function $\mu$ is a **solution of the expression** $BGP(P)$ (where $P$ is a list of triples) if:

- Domain of $\mu$ is the set of variables in $P$
- By replacing bnodes with URIs, bnodes, or RDF literals, $P$ can be transformed into a pattern $P'$ such that:
  All triples in $\mu(P')$ occur in the input graph

**Result of $BGP(P)$:**
List of all such solutions $\mu$ (order undefined)
Unions of Solutions

- Two solutions $\mu$ and $\mu'$ are **compatible** if for all $x$ for which $\mu$ and $\mu'$ are defined we have $\mu(x) = \mu'(x)$

- **Union** of two compatible solutions $\mu$ and $\mu'$:
  - $(\mu \cup \mu')(x) = \mu(x)$ if $\mu$ is defined for $x$
  - $(\mu \cup \mu')(x) = \mu'(x)$ if $\mu'$ is defined for $x$
  - $(\mu \cup \mu')(x) = \text{undefined}$ otherwise

→ simple intuition: union of compatible table rows
(Attention: not related to UNION operator, see below)
Defining SPARQL Operators

Now we can define the essential operations:

- **Filter**\((F, \Psi) = \{\mu \mid \mu \in \Psi \text{ and the expression } \mu(F) \text{ evaluates to } \text{true}\}\)
- **Join**\((\Psi_1, \Psi_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Psi_1, \mu_2 \in \Psi_2, \text{ and } \mu_1 \text{ compatible with } \mu_2\}\)
- **Union**\((\Psi_1, \Psi_2 ) = \{\mu \mid \mu \in \Psi_1 \text{ or } \mu \in \Psi_2 \}\)
- **LeftJoin**\((\Psi_1, \Psi_2, F) = \\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Psi_1, \mu_2 \in \Psi_2, \text{ and } \mu_1 \text{ is compatible to } \mu_2, \text{ and the expression } (\mu_1 \cup \mu_2)(F) \text{ evaluates to } \text{true}\}\) \cup \{\mu_1 \mid \mu_1 \in \Psi_1 \text{ and for all } \mu_2 \in \Psi_2 \text{ we find that: either } \mu_1 \text{ is not compatible with } \mu_2 \text{ or } (\mu_1 \cup \mu_2)(F) \text{ is not } \text{true}\}\)

**Symbols:** \(\Psi, \Psi_1, \Psi_2\) results; \(\mu, \mu_1, \mu_2\) solutions; \(F\) filter condition
Example

@prefix ex: <http://eg.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
ex:Hamlet           ex:author ex:Shakespeare ;
ex:price           "10.50"^^xsd:decimal .
ex:Macbeth          ex:author ex:Shakespeare .
ex:Tamburlaine      ex:author ex:Marlowe ;
ex:price           "17"^^xsd:integer .
ex:DoctorFaustus    ex:author ex:Marlowe ;
ex:price           "12"^^xsd:integer ;
ex:title           "The Tragical History of Doctor Faustus" .
ex:RomeoJuliet      ex:author ex:Brooke ;
ex:price           "9"^^xsd:integer .

  { ?book ex:author ex:Shakespeare . } UNION
  { ?book ex:author ex:Marlowe . } }
Example Calculation (1)

Filter((?price < 15),
  Join(
    LeftJoin(
      true ),
  ) )

→ joins with $Z$ do not affect result and can be removed
Filter((?price < 15),
Join(
  LeftJoin(
    true ),
) )
Example Calculation (3)

Filter((?price < 15),
  Join(
    LeftJoin(
      true ),
    Union( BGP(?book  <http://eg.org/author>
          <http://eg.org/Shakespeare>  .),
          BGP(?book  <http://eg.org/author>
                 <http://eg.org/Marlowe>  .) )
  ) )

<table>
<thead>
<tr>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex:Tamburlaine</td>
</tr>
<tr>
<td>ex:DoctorFaustus</td>
</tr>
</tbody>
</table>
Example Calculation (4)

Filter((?price < 15),
  Join(
    LeftJoin(
      true ),
  )
)

<table>
<thead>
<tr>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex:MacBeth</td>
</tr>
<tr>
<td>ex:Hamlet</td>
</tr>
</tbody>
</table>
Example Calculation (5)

Filter((?price < 15),
  Join(
    LeftJoin(
      true ),
    Union(  BGP (?book  <http://eg.org/author>
                     <http://eg.org/Shakespeare>  .),
            BGP (?book  <http://eg.org/author>
                     <http://eg.org/Marlowe>  .) )
  )
)

<table>
<thead>
<tr>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex:Hamlet</td>
</tr>
<tr>
<td>ex:Tamburlaine</td>
</tr>
<tr>
<td>ex:DoctorFaustus</td>
</tr>
<tr>
<td>ex:MacBeth</td>
</tr>
</tbody>
</table>
Example Calculation (6)

Filter((?price < 15),
Join(
    LeftJoin(
        true ),
    Union( BGP(?book  <http://eg.org/author>
              <http://eg.org/Shakespeare> .),
           BGP(?book  <http://eg.org/author>
              <http://eg.org/Marlowe> .) )
)

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex:Hamlet</td>
<td>&quot;10.50&quot;^^xsd:decimal</td>
</tr>
<tr>
<td>ex:Tamburlaine</td>
<td>&quot;17&quot;^^xsd:integer</td>
</tr>
<tr>
<td>ex:DoctorFaustus</td>
<td>&quot;12&quot;^^xsd:integer</td>
</tr>
<tr>
<td>ex:RomeoJuliet</td>
<td>&quot;9&quot;^^xsd:integer</td>
</tr>
</tbody>
</table>
Example Calculation (7)

Filter((?price < 15),
  Join(  
    LeftJoin( 
      true ),  
    Union( BGP(?book <http://eg.org/author>  
              <http://eg.org/Shakespeare> .),  
          BGP(?book <http://eg.org/author>  
              <http://eg.org/Marlowe> .) )  
  ) )

<table>
<thead>
<tr>
<th>book</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex:DoctorFaustus</td>
<td>&quot;The Tragical ...&quot;</td>
</tr>
</tbody>
</table>
Example Calculation (8)

Filter((?price < 15),
  Join(
    LeftJoin(
      true ),
    Union( BGP(?book <http://eg.org/author>
      <http://eg.org/Shakespeare> .),
      BGP(?book <http://eg.org/author>
        <http://eg.org/Marlowe> .) )
  ) )

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex:Hamlet</td>
<td>&quot;10.50&quot;^^xsd:decimal</td>
<td></td>
</tr>
<tr>
<td>ex:Tamburlaine</td>
<td>&quot;17&quot;^^xsd:integer</td>
<td></td>
</tr>
<tr>
<td>ex:DoctorFaustus</td>
<td>&quot;12&quot;^^xsd:integer</td>
<td>&quot;The Tragical ...&quot;</td>
</tr>
<tr>
<td>ex:RomeoJuliet</td>
<td>&quot;9&quot;^^xsd:integer</td>
<td></td>
</tr>
</tbody>
</table>
Example Calculation (9)

\[
\text{Filter((?price < 15),}
\text{ Join(
\quad \text{LeftJoin(}
\quad \quad \text{BGP(?book <http://eg.org/price> ?price .),}
\quad \quad \text{BGP(?book <http://eg.org/title> ?title .),}
\quad \text{true },
\quad \text{Union( BGP(?book <http://eg.org/author>
\quad \quad \quad <http://eg.org/Shakespeare> .),}
\quad \quad \text{BGP(?book <http://eg.org/author>
\quad \quad \quad <http://eg.org/Marlowe> .) ) })
\]

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex:Hamlet</td>
<td>&quot;10.50&quot;^^xsd:decimal</td>
<td></td>
</tr>
<tr>
<td>ex:Tamburlaine</td>
<td>&quot;17&quot;^^xsd:integer</td>
<td></td>
</tr>
<tr>
<td>ex:DoctorFaustus</td>
<td>&quot;12&quot;^^xsd:integer</td>
<td>&quot;The Tragical ...&quot;</td>
</tr>
</tbody>
</table>
Example Calculation (10)

```
Filter( (?price < 15),
    Join(
        LeftJoin(
            true ),
    )
)
```

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex:Hamlet</td>
<td>&quot;10.50&quot;^^xsd:decimal</td>
<td></td>
</tr>
<tr>
<td>ex:DoctorFaustus</td>
<td>&quot;12&quot;^^xsd:integer</td>
<td>&quot;The Tragical ...&quot;</td>
</tr>
</tbody>
</table>
Problem: “Given an input graph, a SPARQL query, and a potential solution, find out whether the solution is correct.”

How hard is this, computationally speaking?

- It’s certainly as hard as Sudoku or Minesweeper (NP)
- It’s also as hard as Reversi or Sokoban (PSPACE) (needs Union & Optional, but no Filters)
- But it’s not as hard as Draughts or Chess (EXPTime)

SPARQL is indeed PSpace-complete (recall: OWL 2 profiles→PTIME, RDF(S) entailment→NP, OWL 2 DL→N2EXPTime)

Word of Warning: generalised games make nice memory hooks, but human intuition may still be very wrong about algorithmic hardness.

*) Homework: find out how to solve a given Sudoku with SPARQL; try doing it without Filters, Unions, or Optional
Negation in SPARQL

Can we query negative information with SPARQL?

- RDF knows no negation
- SPARQL has no negation operator, but inequality and ! in filters
- Yet “negation as failure” can be implemented using \texttt{OPTIONAL}, Boolean negation, and \texttt{BOUND}:

```sparql
PREFIX ex: <http://example.org/>
SELECT ?book
WHERE
  OPTIONAL { ?book ex:editor ?editor } 
  FILTER( !BOUND(?editor) ) }
```

This is the only use for \texttt{BOUND}! (better syntax in next SPARQL?!)
SPARQL fully defined only for plain RDF → what about RDFS and OWL?

- SPARQL with a model-theoretic twist: “Which graph patterns occur in all models?”
- Semantics of basic patterns and unions clear (variables as subjects/objects only)
  → Closely related to “conjunctive queries”
- Semantics for optional patterns and filters not so clear
- Returning unnamed individuals as blank nodes could lead to infinite results in OWL – treat input bnodes like constants?
- Even if semantics is clear, it may not be implementable: Is SPARQL for OWL DL decidable? (open problem)
- Very high worst-case complexities expected (above 2ExpTime)

→ Current implementations support subsets of SPARQL for OWL DL
The Future of SPARQL

The new SPARQL working group will define a compatible extension of SPARQL.

Considered features:

- Extended query language:
  - Aggregate functions
  - Subqueries
  - Negation (i.e. a better syntax for `OPTIONAL + BOUND`)
- Service description (report service's query capabilities)
- Definition of semantics for RDF(S), OWL (hopefully)
- Further features as time permits …
Summary and Outlook

SPARQL: a query language for RDF

• W3C standard, widely used
• Queries based on graph patterns
• Various extensions (filters, modifiers, result formats)
• Semantics via translation to SPARQL algebra
• Extension to OWL possible but not trivial
• Update of SPARQL specification in progress
Further Reading

- P. Hitzler, S. Rudolph, M. Krötzsch: **Foundations of Semantic Web Technologies.** CRC Press, 2009. (Chapter 7 closely related to this lecture)
- E. Prud'hommeaux, A. Seaborne: **SPARQL query language for RDF.** See [http://www.w3.org/TR/rdf-sparql-query/](http://www.w3.org/TR/rdf-sparql-query/) W3C Recommendation, Jan 15 2008. (the official standard)

Selected research articles: