IN3060/4060 – Semantic Technologies – Spring 2021

Lecture 4: The SPARQL Query Language

Jieying Chen

5th February 2021

Department of Informatics

University of Oslo

Mandatory exercises

- Remember: Hand-in Oblig 2 by today.

- Oblig 3, SPARQL, is published after this lecture.
- Hand-in by Friday next week.
- Use Mr. Oblig to test your solutions.

Today's Plan

Outline

SPARQL

- SPARQL Protocol And RDF Query Language
- Standard language to query graph data represented as **RDF triples**
- W3C Recommendations
  - **SPARQL 1.0:** W3C Recommendation 15 January 2008
  - **SPARQL 1.1:** W3C Recommendation 21 March 2013
- This lecture is about SPARQL 1.0.
- Documentation:
  - Syntax and semantics of the SPARQL query language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`

Outline

Recap: RDF triples

- The W3C representation of knowledge in the Semantic Web is RDF (Resource Description Framework)
- RDF talks about *resources* identified by URIs.
- In RDF, all knowledge is represented by *triples* (aka statements or facts)
- A triple consists of *subject*, *predicate*, and *object*
- The *subject* maybe a resource or a blank node
- The *predicate* must be a resource
- The *object* can be a resource, a blank node, or a literal

Recap: RDF Literals

- Can only appear as object in the *object* in the triple.
- Literals can be
  - Plain, without language tag:

    ```
    geo:berlin geo:name "Berlin" .
    ```
  - Plain, with language tag:

    ```
    geo:germany geo:name "Deutschland"@de .
    geo:germany geo:name "Germany"@en .
    ```
  - Typed, with a URI indicating the type:

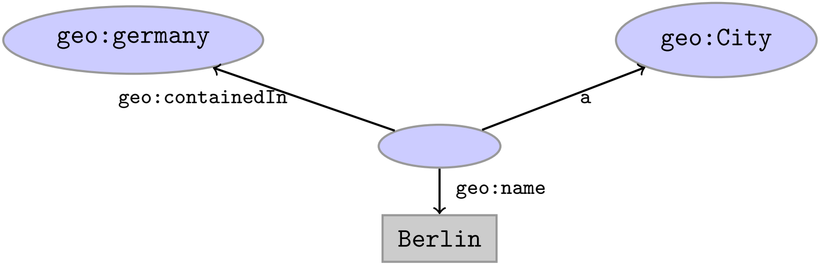
    ```
    geo:berlin geo:population "3431700"^^xsd:integer .
    ```

Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a city in Germany called Berlin

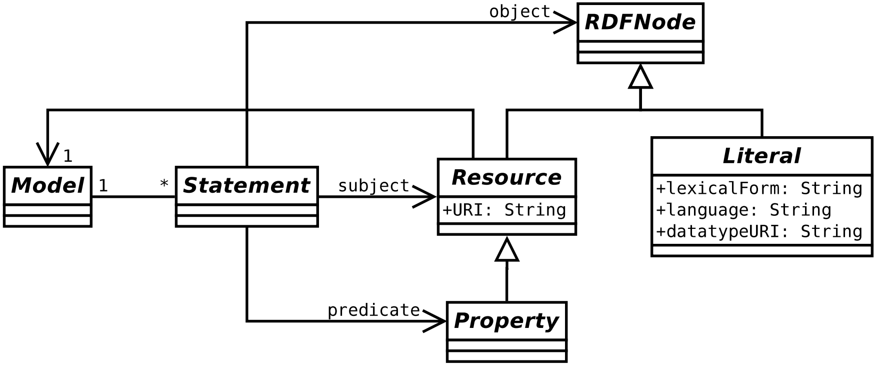
```
_:x a geo:City .
_:x geo:containedIn geo:germany .
_:x geo:name "Berlin" .
```



Recap: Jena

- Jena is a Semantic Web programming framework for Java.
- Open source.
- API to extract data from and write to RDF graphs.
- Includes an engine to query RDF graphs through SPARQL.
- Interfaces for main RDF elements `Resource`, `Property`, `Literal`, `Statement`, `Model`
- The RDF graphs are represented as an abstract `Model`.

Recap: Jena


object
RDFNode
Model
1
1
*
Statement
subject
Resource
+URI: String
Literal
+lexicalForm: String
+language: String
+datatypeURI: String
predicate
Property

Recap: Vocabularies

- Best Practices: Reuse vocabularies to ease interoperability.
  - People are more familiar with them
  - Can be queried more easily
  - The semantics must be clear, shouldn't twist the meaning too much.
- Good starting point:
  - Linked Open Vocabularies: `http://lov.okfn.org/`
  - Schema.org: `https://schema.org`

Recap: RDF and RDFS Vocabularies

- Prefix `rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
- Prefix `rdfs:<http://www.w3.org/2000/01/rdf-schema#>`
- They need to be declared like all others.
- Examples:

```
geo:berlin rdf:type geo:City .
geo:containedIn a rdf:Property .
geo:berlin rdfs:label geo:City .
```

- Note that the keyword “a” is an alternative for `rdf:type`.

Recap: Friend Of A Friend

- People, personal information, friends, see `http://www.foaf-project.org/`
- Prefix `foaf:<http://xmlns.com/foaf/0.1/>`
- Important elements:

Person	a person, alive, dead, real, imaginary
name	name of a person (also firstName, familyName)
mbox	mailbox URL of a person
knows	a person knows another

- Examples:

```
<https://w3id.org/scholarlydata/person/ernesto-jimenez-ruiz>
      a foaf:Person ;
      foaf:name "Ernesto Jiménez-Ruiz" ;
      foaf:mbox <mailto:ernestoj@ifi.uio.no> ;
      foaf:knows <http://heim.ifi.uio.no/martingi/foaf#me> .
```

Recap: Dublin Core

- Metadata for documents, see `http://dublincore.org/`.
- Prefix `dc:<http://purl.org/dc/terms/>`
- Important elements:

creator	a document's main author
created	the creation date
title	title of document
description	a natural language description

- Examples:

```
<https://w3id.org/scholarlydata/.../iswc2016/paper/research/res
      dc:creator
<https://w3id.org/scholarlydata/person/ernesto-jimenez-ruiz>;
      dc:created "2016-10-20" ;
      dc:description "ISWC research paper number 146"@en ;
```

Outline

SPARQL by Example

- SPARQL Protocol And RDF Query Language
- Try it out:

	`https://www.w3.org/wiki/SparqlEndpoints`
DBpedia	`http://dbpedia.org/sparql`
Wikidata	`https://query.wikidata.org/`
Musicbrainz	`http://dbtune.org/musicbrainz/snorql/`
EBI	`https://www.ebi.ac.uk/rdf/`

Simple Examples

- DBpedia information about actors, movies, etc.
  `https://dbpedia.org/`
- Web interface for SPARQL writing: `http://dbpedia.org/sparql`

People called "Johnny Depp"

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?jd WHERE {
    ?jd foaf:name "Johnny Depp"@en .
}
```

Answer:

?jd
<http://dbpedia.org/resource/Johnny_Depp>

Simple Examples (cont.)

Films starring people called "Johnny Depp"

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX dbo:  <http://dbpedia.org/ontology/>
SELECT ?m WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
}
```

Answer:

?m
<http://dbpedia.org/resource/Dead_Man>
<http://dbpedia.org/resource/Edward_Scissorhands>
<http://dbpedia.org/resource/Arizona_Dream>
...

Simple Examples (cont.)

Titles of films by people called “Johnny Depp”

```
SELECT ?title WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
    ?m rdfs:label ?title .
}
```

Answer:

?title
"Truposz"@pl
"Dead Man"@en
"El sueño de Arizona"@es
"Arizona Dream"@en
...

Simple Examples (cont.)

Names of people who co-starred with “Johnny Depp”

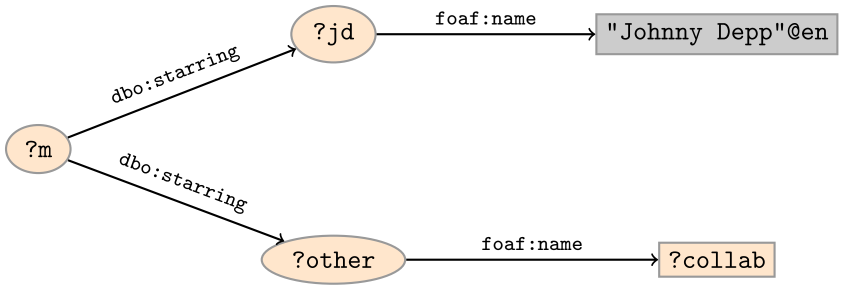
```
SELECT DISTINCT ?collab WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
    ?m dbo:starring ?other .
    ?other foaf:name ?collab .
}
```

Answer:

?collab
"Al Pacino"@en
"Antonio Banderas"@en
"Johnny Depp"@en
"Marlon Brando"@en
...

Graph Patterns

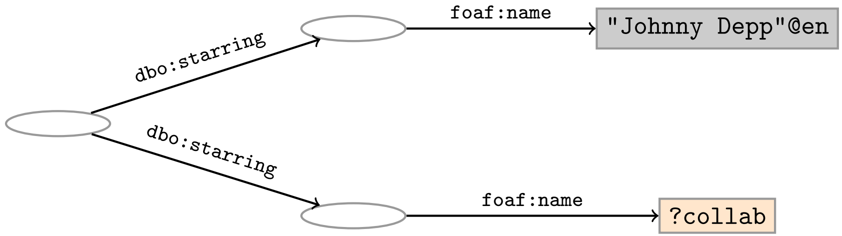
The previous SPARQL query as a graph:



Pattern matching: assign values to variables to make this a sub-graph of the RDF graph!

Graph with blank nodes

Variables not SELECTed can equivalently be blank:



Pattern matching: assign values to variables and blank nodes to make this a sub-graph of the RDF graph!

SPARQL Query with blank nodes

Names of people who co-starred with “Johnny Depp”

```
SELECT DISTINCT ?collab WHERE {
    _:jd foaf:name "Johnny Depp"@en .
    _:m dbo:starring _:jd .
    _:m dbo:starring _:other .
    _:other foaf:name ?collab .
}
```

The same with blank node syntax

```
SELECT DISTINCT ?collab WHERE {
    [ dbo:starring [foaf:name "Johnny Depp"@en] ,
                   [foaf:name ?collab]
    ]
}
```

Outline

Components of an SPARQL query

Prologue: prefix definitions Results form specification: (1) variable list, (2) type of query (SELECT, ASK, CONSTRUCT, DESCRIBE), (3) remove duplicates (DISTINCT, REDUCED) Dataset specification Query pattern: graph pattern to be matched Solution modifiers: ORDER BY, LIMIT, OFFSET

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX dbo:   <http://dbpedia.org/ontology/>
SELECT DISTINCT ?collab
FROM <http://dbpedia_dataset>
WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?pub dbo:starring ?jd .
    ?pub dbo:starring ?other .
    ?other foaf:name ?collab .
    FILTER (STR(?collab)!="Johnny Depp"@en)
}
```

Types of Queries

SELECT Compute table of bindings for variables

```
SELECT ?a ?b WHERE {
  [ dbo:starring ?a ;
    dbo:starring ?b ]
}
```

CONSTRUCT Use bindings to construct a new RDF graph

```
CONSTRUCT {
  ?a foaf:knows ?b .
} WHERE {
  [ dbo:starring ?a ;
    dbo:starring ?b ]
}
```

Types of Queries (cont.)

ASK Answer (yes/no) whether there is ≥ 1 match

```
ASK WHERE {
    ?jd foaf:name "Johnny Depp"@en .
}
```

DESCRIBE Returns and RDF graph with data about matching resources

```
DESCRIBE ?jd WHERE {
    ?jd foaf:name "Johnny Depp"@en .
}
```

Solution Sequences and Modifiers

- Permitted to SELECT queries only
- SELECT treats solutions as a sequence (solution sequence)
- Query patterns generate an unordered collection of solutions
- *Sequence modifiers* can modify the solution sequence (not the solution itself):
  - Order
  - Projection
  - Distinct
  - Reduced
  - Offset
  - Limit
- **Applied in this order.**

ORDER BY

- Used to sort the solution sequence in a given way:
- `SELECT ... WHERE ... ORDER BY ...`
- ASC for ascending order (default) and DESC for descending order
- E.g.

```
SELECT ?city ?pop WHERE {
  ?city geo:containedIn ?country ;
        geo:population ?pop .
} ORDER BY ?country ?city DESC(?pop)
```

- Standard defines sorting conventions for literals, URIs, etc.
- Not all “sorting” variables are required to appear in the solution

Projection, DISTINCT, REDUCED

- Projection means that only some variables are part of the solution
  - Done with `SELECT ?x ?y WHERE {?x ?y ?z...}`
- DISTINCT eliminates (all) duplicate solutions:
  - Done with `SELECT DISTINCT ?x ?y WHERE {?x ?y ?z...}`
  - A solution is a duplicate if it assigns the same RDF terms to all selected variables as another solution.
- REDUCED allows to remove *some* or all duplicate solutions
  - Done with `SELECT REDUCED ?x ?y WHERE {?x ?y ?z...}`
  - Motivation: Can be expensive to find and remove all duplicates
  - Leaves amount of removal to implementation (e.g. consecutive occurrences)
  - Rarely used...

OFFSET and LIMIT

- LIMIT: limits the number of results
- OFFSET: position/index of the first returned result
- Useful for paging through a large set of solutions
- ...but not useful for implementing paging in applications.
- Can compute solutions number 51 to 60
- Done with
  ```
  SELECT ... WHERE {...} ORDER BY ...
  LIMIT 10 OFFSET 50
  ```
- LIMIT and OFFSET can be used separately
- `OFFSET` not meaningful without `ORDER BY`.

Query patterns

- Different types of *graph patterns* for the query pattern (WHERE clause):
    - Basic Graph Patterns (BGP)
    - Group Graph Patterns
    - Filters or Constraints (FILTER)
    - Optional Graph Patterns (OPTIONAL)
    - Union Graph Patterns (UNION, Matching Alternatives)
    - Graph Graph Patterns (RDF Datasets)

Basic Graph Patterns (BGP)

- A *Basic Graph Pattern* is a set of triple patterns.
- e.g.

```
?jd foaf:name "Johnny Depp"@en .
_:m dbo:starring ?jd .
_:m dbo:starring ?other .
```

- Scope of blank node labels is the BGP
- Basically: A *match* is a function that maps
  - every variable and every blank node in the pattern
  - to a resource, a blank node, or a literal in the RDF graph (an “RDF term”)

Group Graph Patterns

- Group several patterns with { and }.
- A group containing *one* basic graph pattern:

```
{
  _:m dbo:starring ?jd .
  _:m dbo:starring ?other .
}
```

- Two groups with one basic graph pattern each:

```
{
  { _:m1 dbo:starring ?jd . }
  { _:m2 dbo:starring ?other . }
}
```

- Note: Same name for two different blank nodes not allowed!
- The scope of a FILTER constraint is the group where the filter appears.

Filters

- Groups may include *constraints* or *filters*
- Reduces matches of surrounding group where filter applies
- E.g.

```
{
  ?x a dbo:Place ;
     dbpprop:population ?pop .
  FILTER (?pop > 1000000)
}
```

- E.g.

```
{
  ?x a dbo:Document ;
     dbpprop:abstract ?abs .
  FILTER (lang(?abs) = "no")
}
```

Filters: Functions and Operators

- Usual binary operators: `||`, `&&`, `=`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`.
- Usual unary operators: `!`, `+`, `-`.
- Unary tests: `bound(?var)`, `isURI(?var)`, `isBlank(?var)`, `isLiteral(?var)`.
- Accessors: `str(?var)`, `lang(?var)`, `datatype(?var)`
- `regex` is used to match a variable with a regular expression. *Always use with* `str(?var)`. E.g.: `regex(str(?name), "Oslo")`.

Read the spec for details!

Optional Patterns

- Allows a match to leave some variables *unbound* (e.g. no data was available)
- A *partial* function from variables to RDF terms
- Groups may include *optional parts*
- E.g.

```
{
  ?x a dbo:Document ;
     dbp:date ?date .
  OPTIONAL {
    ?x dbp:abstract ?abs .
    FILTER (lang(?abs) = "no")
  }
}
```

- `?x` and `?date` bound in every match, `?abs` bound if there is a Norwegian abstract
- Groups can contain several optional parts, evaluated separately

Optional Patterns: Negation as Failure

- Testing if a graph pattern is not expressed. . .
- . . . by specifying an OPTIONAL graph pattern that introduces a variable,
- and testing if the variable is not bound.
- E.g.

```
{
  ?x foaf:givenName ?name .
  OPTIONAL {
    ?x dc:date ?date .
    FILTER (!bound(?date))
  }
}
```

- Called **Negation as Failure** in logic programming

Matching Alternatives (UNION)

- A UNION pattern matches if any of some alternatives matches
- E.g.

```
{
  { ?book dbo:starring ?author ;
          dc:created ?date . }
  UNION
  { ?book foaf:maker ?author . }
  UNION
  { ?author foaf:made ?book . }
}
```

Graph Graph Patterns (RDF datasets)

- SPARQL queries are executed against an **RDF dataset**
- An RDF dataset comprises
  - One **default graph** (unnamed) graph.
  - Zero or more **named graphs** identified by an URI
- FROM and FROM NAMED keywords allows to select an RDF dataset by reference
  - The **default graph** will consist of the RDF merge of the graphs referred to in the FROM clauses,
  - FROM NAMED clauses will define the different named graphs.
  - Note that, if there is no FROM clause, but there are FROM NAMED clauses, the default graph will be empty.
- Keyword GRAPH makes the named graphs the **active graph** for pattern matching
  - A specific (named) graph can be used as active graph if its IRI is provided.

Default graph example

Add three RDF datasets to default graph

```
SELECT ?kname ?fname
FROM <http://data.lenka.no/dumps/fylke-geonames.ttl>
FROM <http://data.lenka.no/dumps/kommune-navn.ttl>
FROM <http:// .../dumps/kommunesentre-geonames.ttl>
WHERE {
  ?fylke a gd:Fylke ;
         gn:officialName ?fname ;
         gn:childrenFeatures ?kommune .
  ?kommune a gd:Kommune ;
           gn:officialName ?kname ;
  FILTER (langMatches(lang(?fname), 'no'))
  FILTER (langMatches(lang(?kname), 'no'))
}
```

Named graph example 1

Occurrences of Bob in different datasets

```
SELECT ?iri_graph ?bobNick
FROM NAMED <http://example.org/foaf/aliceFoaf>
FROM NAMED <http://example.org/foaf/bobFoaf>
WHERE {
  {
    GRAPH ?iri_graph {
      ?x foaf:mbox <mailto:bob@work.example> .
      ?x foaf:nick ?bobNick .
    }
  }
```

Named graph example 2

Take coordinates from one source only

```
SELECT *
FROM <http://data.lenka.no/dumps/kommune-navn.ttl>
FROM <http://data.lenka.no/dumps/kommunesentre-geonames.ttl>
FROM NAMED <http://data.lenka.no/dumps/kommunesentre-geonames.ttl>
FROM NAMED <http://sws.geonames.org/6453350/about.rdf>
WHERE {
  ?feature gn:officialName "Lillehammer"@no .
  OPTIONAL {
    GRAPH <http://data.lenka.no/dumps/kommunesentre-geonames.ttl> {
      ?feature pos:lat ?lat ;
               pos:long ?long ;
    }
  }
}
```

Outline

SPARQL in Jena

- SPARQL functionality bundled with Jena has separate Javadocs:

  `http://jena.apache.org/documentation/javadoc/arq/`
- Main classes in package `org.apache.jena.query`
  - `Query` a SPARQL query
  - `QueryFactory` for creating queries in various ways
  - `QueryExecution` for the execution state of a query
  - `QueryExecutionFactory` for creating query executions (to get `QueryExecution` instances)
  - `DatasetFactory` for creating dataset instances
  - For `SELECT` queries:
    - `QuerySolution`, a single solution to the query.
    - `ResultSet`, all the `QuerySolutions` (an iterator)
    - `ResultSetFormatter`, turn a `ResultSet` into various forms: text, RDF graph (Model, in Jena terminology) or plain XML
  - `CONSTRUCT` and `DESCRIBE` return `Models`, `ASK` a Java boolean.

Constructing a Query and a QueryExecution

- Query objects are usually constructed by parsing:

```
String qStr =
        "PREFIX foaf:  <" + foafNS + ">"
    + "SELECT ?a ?b WHERE {"
    + "  ?a foaf:knows ?b ."
    + "} ORDER BY ?a ?b";
Query q = QueryFactory.create(qStr);
```

- A `Query` can be used several times, on multiple models
- For each execution, a new `QueryExecution` is needed
- To produce a `QueryExecution` for a given `Query` and `Model`:

```
QueryExecution qe =
    QueryExecutionFactory.create(q, model);
```

Executing a Query

- `QueryExecution` contains methods to execute different kinds of queries (`SELECT`, `CONSTRUCT`, etc.)
- E.g. for a `SELECT` query:
  ```
  ResultSet res = qe.execSelect();
  ```
- E.g. for a `CONSTRUCT` query:
  ```
  Model construct_model = qe.execConstruct();
  ```
- `ResultSet` is a sub-interface of `Iterator<QuerySolution>`
- `QuerySolution` has methods to get list of variables, value of single variables, etc.
- Important to call `close()` on query executions when no longer needed.

Example: SPARQL in Jena

```
String qStr = "SELECT ?a ?b ...";
Query q = QueryFactory.create(qStr);

QueryExecution qe =
   QueryExecutionFactory.create(q, model);

try {
   ResultSet res = qe.execSelect();
   while( res.hasNext()) {
      QuerySolution soln = res.next();
      RDFNode a = soln.get("?a");
      RDFNode b = soln.get("?b");
      System.out.println(""+a+" knows "+b);
   }
} finally {
   qe.close();
}
```

Querying a Model, Dataset or Endpoint

- Querying a model:

```
     Model model = ModelFactory.createDefaultModel();
     model.read("http://heim.ifi.uio.no/martingi/foaf");
     QueryExecutionFactory.create(q, model);
```

- Querying a Dataset:

```
     String dftGraphURI =
"http://heim.ifi.uio.no/martingi/foaf" ;
     List namedGraphURIs = new ArrayList() ;

namedGraphURIs.add("http://richard.cyganiak.de/foaf.rdf");
     namedGraphURIs.add("http://danbri.org/foaf.rdf");
     Dataset dataset = DatasetFactory.create(dftGraphURI,
namedGraphURIs);
     QueryExecutionFactory.create(q, dataset);
```

Querying a Model, Dataset or Endpoint (cont.)

- Jena can also send SPARQL queries to a remote endpoint!
  - Use `sparqlService` in `QueryExecutionFactory`
  - E.g.

```
String endpoint = "http://dblp.l3s.de/d2r/sparql";
String qStr = "SELECT ?a ?b ...";
Query q = QueryFactory.create(qStr);

QueryExecution qe =
   QueryExecutionFactory.sparqlService(endpoint,q);

try {
   ResultSet res = qe.execSelect();
   ...
} finally {
   qe.close();
}
```

SPARQL Injection

- Antipattern:
  `String sStr = "SELECT..." + name + "...";`
  where `name` comes from user input
- Tricky content of `name` can be a security issue!
- Have to be careful to escape content of `name` properly
- Best to use “parameterised SPARQL strings”
  `https://jena.apache.org/documentation/query/parameterized-sparql-strings.html`

SPARQL on the 'Net

- Many sites (Wikidata, dbpedia, dbtunes,. . .) publish *SPARQL endpoints*
- I.e. SPARQL queries can be submitted to a database server that sends back the results
- Uses HTTP to submit URL-encoded queries to server
  `GET /sparql/?query=...  HTTP/1.1`
- Actually defined via W3C Web Services, see

  `http://www.w3.org/TR/rdf-sparql-protocol/`
- Try it out:

	`https://www.w3.org/wiki/SparqlEndpoints`
DBpedia	`http://dbpedia.org/sparql`
Wikidata	`https://query.wikidata.org/`
Musicbrainz	`http://dbtune.org/musicbrainz/snorql/`
EBI	`https://www.ebi.ac.uk/rdf/`

Outline

Wrap-up

- SPARQL is a W3C-standardised query language for RDF graphs
- It is built around "graph patterns"
- Comes with a protocol to communicate with "endpoints"
- Can be conveniently used with Jena and tens of other systems.

More to come: SPARQL 1.1

SPARQL 1.1 became W3C Recommendations 21 March 2013.

- Updates (add/delete triples)
- Service Descriptions
- Basic Federated query
- Subqueries.
- Property paths (to shorten common queries)
- Aggregate functions (count, sum, average,. . .)
- Negation, set difference, i.e. something is *not* in a graph
- Entailment regimes

Additional material

An Introduction to SPARQL by Olaf Hartig: `http://www.slideshare.net/olafhartig/an-introduction-to-sparql`

SPARQL Query Language for RDF (SPARQL 1.0 W3C Recommendation): `https://www.w3.org/TR/rdf-sparql-query/`