SPARQL Query Language

Ernesto Jiménez-Ruiz
Lecturer in Artificial Intelligence

Polls and Forums

- Group work VS individual work.
- Extra lab hour during during drop-in session (Wednesdays 1-2pm).
- Reading week (Week 6) extra session for Q&A.
- Please do not hesitate to use the forums.

Recap: RDF-based Knowledge Graphs

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)

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*

	s	p	o
• URI references may occur in all positions	✔	✔	✔
• Literals may only occur in object position	✘	✘	✔
• Blank nodes can not occur in predicate position	✔	✘	✔

Recap: RDF Literals

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

```
dbp:london rdfs:label "London" .
```

  - Plain, with language tag:

```
dbp:london rdfs:label "Londres"@es .
dbp:london rdfs:label "London"@en .
```

  - Typed, with a URI indicating the type:

```
dbp:london dbpo:population 9,304,000^^xsd:integer .
```

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:

```
dbp:london rdf:type geo:City .
dbpo:locationCountry a rdf:Property .
dbp:london rdfs:label "London" .
```

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

Recap: RDF Example

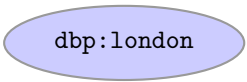
London is a city in England called Londres in Spanish

```
dbp:london a dbpo:City .
dbp:london dbpo:locationCountry dbp:england .
dbp:london rdfs:label "Londres"@es .
```

Recap: RDF Example

London is a city in England called Londres in Spanish

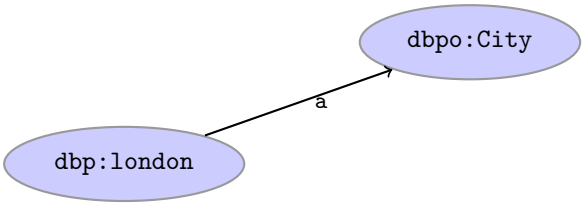
```
dbp:london a dbpo:City .
dbp:london dbpo:locationCountry dbp:england .
dbp:london rdfs:label "Londres"@es .
```



Recap: RDF Example

London is a city in England called Londres in Spanish

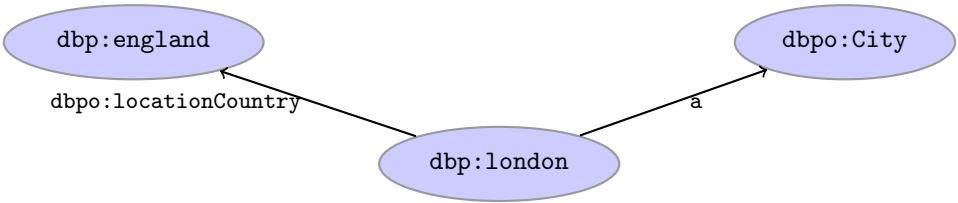
```
dbp:london a dbpo:City .
dbp:london dbpo:locationCountry dbp:england .
dbp:london rdfs:label "Londres"@es .
```



Recap: RDF Example

London is a city in England called Londres in Spanish

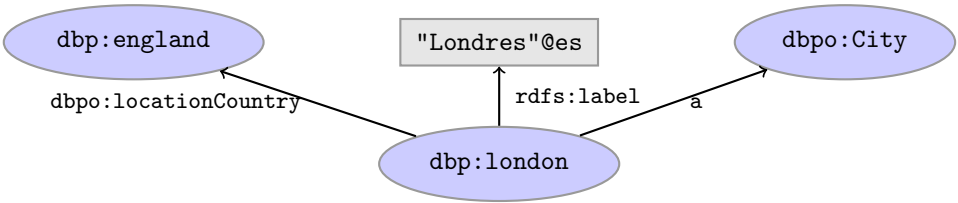
```
dbp:london a dbpo:City .
dbp:london dbpo:locationCountry dbp:england .
dbp:london rdfs:label "Londres"@es .
```



Recap: RDF Example

London is a city in England called Londres in Spanish

```
dbp:london a dbpo:City .
dbp:london dbpo:locationCountry dbp:england .
dbp:london rdfs:label "Londres"@es .
```



Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2021 with code INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbpo:year "2021"^^xsd:gYear .
_:x city:code "INM713" .
```

Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2021 with code INM713

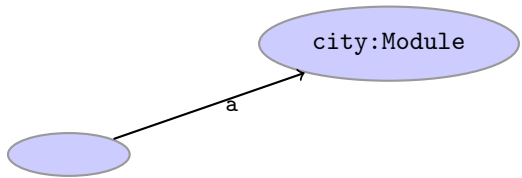
```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbpo:year "2021"^^xsd:gYear .
_:x city:code "INM713" .
```

Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2021 with code INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbpo:year "2021"^^xsd:gYear .
_:x city:code "INM713" .
```

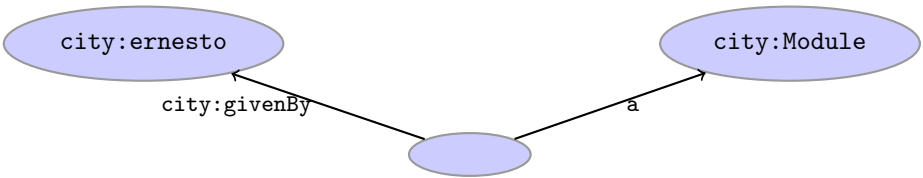


Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2021 with code INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbpo:year "2021"^^xsd:gYear .
_:x city:code "INM713" .
```

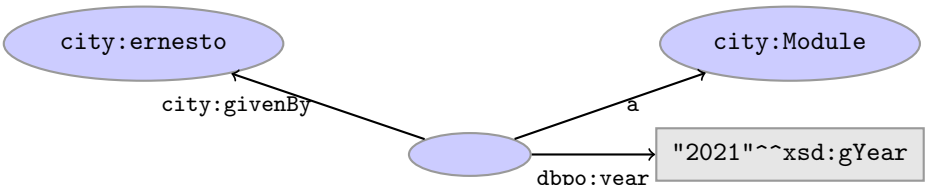


Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2021 with code INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbpo:year "2021"^^xsd:gYear .
_:x city:code "INM713" .
```

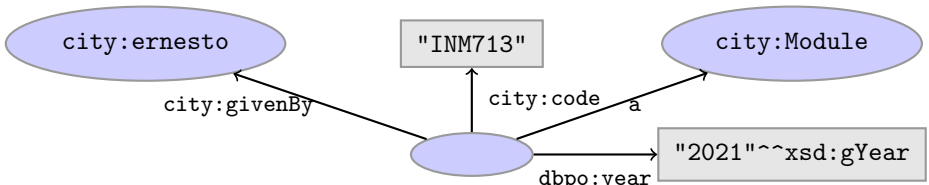


Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2021 with code INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbpo:year "2021"^^xsd:gYear .
_:x city:code "INM713" .
```



Recap: RDF in Python with RDFLib (i)

– We rely on RDFLib: `from rdflib import Graph`

Recap: RDF in Python with RDFLib (i)

- We rely on RDFLib: `from rdflib import Graph`
- Creates empty graph: `g = Graph()`

Recap: RDF in Python with RDFLib (i)

- We rely on RDFLib: `from rdflib import Graph`
- Creates empty graph: `g = Graph()`
- Loads an RDF graph: `g.parse("beatles.ttl", format="ttl")`

Recap: RDF in Python with RDFLib (i)

- We rely on RDFLib: `from rdflib import Graph`
- Creates empty graph: `g = Graph()`
- Loads an RDF graph: `g.parse("beatles.ttl", format="ttl")`
- Saves and RDF graph:
  `g.serialize(destination='beatles.rdf', format='xml')`

Recap: RDF in Python with RDFLib (i)

- We rely on RDFLib: `from rdflib import Graph`
- Creates empty graph: `g = Graph()`
- Loads an RDF graph: `g.parse("beatles.ttl", format="ttl")`
- Saves and RDF graph:
  `g.serialize(destination='beatles.rdf', format='xml')`
- Iterates over a graph:

```
for s, p, o in g:
    print((s.n3(), p.n3(), o.n3()))
```

Recap: RDF in Python with RDFLib (ii)

– Basic triple elements: `from rdflib import URIRef, BNode, Literal`

Recap: RDF in Python with RDFLib (ii)

- Basic triple elements: `from rdflib import URIRef, BNode, Literal`
- Creates an URI:

```
ernesto = URIRef("http://ex.org/univ/city#ernesto")
```

Recap: RDF in Python with RDFLib (ii)

- Basic triple elements: `from rdflib import URIRef, BNode, Literal`
- Creates an URI:
  `ernesto = URIRef("http://ex.org/univ/city#ernesto")`
- Creates a blank node: `bnode = BNode()`

Recap: RDF in Python with RDFLib (ii)

- Basic triple elements: `from rdflib import URIRef, BNode, Literal`
- Creates an URI:
  `ernesto = URIRef("http://ex.org/univ/city#ernesto")`
- Creates a blank node: `bnode = BNode()`
- Creates a literal: `year = Literal('2021', datatype=XSD.gYear)`

Recap: RDF in Python with RDFLib (iii)

– Namespaces:

```
from rdflib import Namespace
```

Recap: RDF in Python with RDFLib (iii)

- Namespaces:
  `from rdflib import Namespace`
- Default namespaces and vocabulary: `from rdflib.namespace import OWL, RDF, RDFS, FOAF, XSD`

Recap: RDF in Python with RDFLib (iii)

- Namespaces:
  `from rdflib import Namespace`
- Default namespaces and vocabulary: `from rdflib.namespace import OWL, RDF, RDFS, FOAF, XSD`
  - *e.g.*, RDF.type is equivalent to
    `URIRef("http://www.w3.org/1999/02/22-rdf-syntax-ns#type")`

Recap: RDF in Python with RDFLib (iii)

- Namespaces:
  `from rdflib import Namespace`
- Default namespaces and vocabulary: `from rdflib.namespace import OWL, RDF, RDFS, FOAF, XSD`
  - *e.g.*, RDF.type is equivalent to
    `URIRef("http://www.w3.org/1999/02/22-rdf-syntax-ns#type")`
- User defined:
  `city = Namespace("http://ex.org/univ/city#")`

Recap: RDF in Python with RDFLib (iii)

- Namespaces:
  `from rdflib import Namespace`
- Default namespaces and vocabulary: `from rdflib.namespace import OWL, RDF, RDFS, FOAF, XSD`
  - *e.g.*, RDF.type is equivalent to
    `URIRef("http://www.w3.org/1999/02/22-rdf-syntax-ns#type")`
- User defined:
  `city = Namespace("http://ex.org/univ/city#")`
  - *e.g.*, `city.ernesto` is equivalent to
    `URIRef("http://ex.org/univ/city#ernesto")`

Recap: RDF in Python with RDFLib (iv)

– Adding triples:

```
g.add((city.ernesto, RDF.type, FOAF.Person))
g.add((city.ernesto, FOAF.name, name))
g.add((city.ernesto, city.teaches, city.inm713))
```

Recap: RDF in Python with RDFLib (iv)

- Adding triples:

```
g.add((city.ernesto, RDF.type, FOAF.Person))
g.add((city.ernesto, FOAF.name, name))
g.add((city.ernesto, city.teaches, city.inm713))
```

- Prefixes: `g.bind("city", city)`

Recap: RDF in Java with Jena API (i)

– Creates empty graph:

```
Model model = ModelFactory.createDefaultModel();
```

Recap: RDF in Java with Jena API (i)

– Creates empty graph:

```
Model model = ModelFactory.createDefaultModel();
```

– Loads an RDF graph:

```
Dataset dataset = RDFDataMgr.loadDataset(file);
Model model = dataset.getDefaultModel();
```

Recap: RDF in Java with Jena API (i)

- Creates empty graph:

```
Model model = ModelFactory.createDefaultModel();
```

- Loads an RDF graph:

```
Dataset dataset = RDFDataMgr.loadDataset(file);
Model model = dataset.getDefaultModel();
```

- Saves and RDF graph:

```
OutputStream out = new FileOutputStream(output_file);
RDFDataMgr.write(out, model, RDFFormat.TURTLE);
```

Recap: RDF in Java with Jena API (i)

- Creates empty graph:

```
Model model = ModelFactory.createDefaultModel();
```

- Loads an RDF graph:

```
Dataset dataset = RDFDataMgr.loadDataset(file);
Model model = dataset.getDefaultModel();
```

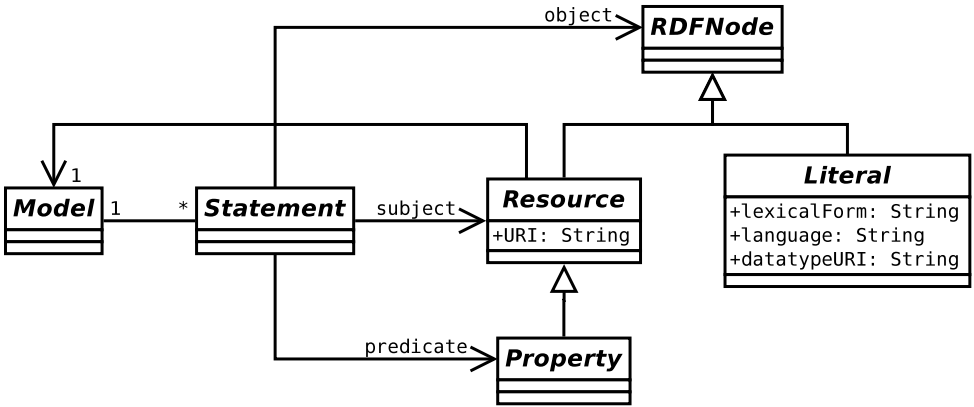
- Saves and RDF graph:

```
OutputStream out = new FileOutputStream(output_file);
RDFDataMgr.write(out, model, RDFFormat.TURTLE);
```

- Iterator over RDF statements:

```
StmtIterator iter = model.listStatements();
```

Recap: RDF in Java (ii)



Recap: RDF in Java (iii)

– Creates a Resource:

```
Resource ernesto_res =
model.createResource("http://ex.org/univ/city#ernesto");
```

Recap: RDF in Java (iii)

- Creates a Resource:

```
Resource ernesto_res =
model.createResource("http://ex.org/univ/city#ernesto");
```

- Creates a blank node: `Resource blank = model.createResource();`

Recap: RDF in Java (iii)

- Creates a Resource:

```
Resource ernesto_res =
model.createResource("http://ex.org/univ/city#ernesto");
```

- Creates a blank node: `Resource blank = model.createResource();`

- Creates an Property:

```
Property teaches_prop =
model.createProperty("http://ex.org/univ/city#teaches");
```

Recap: RDF in Java (iii)

- Creates a Resource:

```
Resource ernesto_res =
model.createResource("http://ex.org/univ/city#ernesto");
```

- Creates a blank node: `Resource blank = model.createResource();`

- Creates an Property:

```
Property teaches_prop =
model.createProperty("http://ex.org/univ/city#teaches");
```

- Creates a literal: `Literal year_lit =`
  `model.createTypedLiteral("2021", XSDDatatype.XSDgYear);`

Recap: RDF in Java (iv)

– Default namespaces and vocabulary:

```
import org.apache.jena.datatypes.xsd.XSDDatatype;
import org.apache.jena.vocabulary.RDF;
```

Recap: RDF in Java (iv)

- Default namespaces and vocabulary:

```
import org.apache.jena.datatypes.xsd.XSDDatatype;
import org.apache.jena.vocabulary.RDF;
```

  - *e.g.*, RDF.type is equivalent to

```
model.createProperty(
"http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
);
```

Recap: RDF in Java (v)

– Adding triples:

```
model.add(ernesto_res, RDF.type, Person_res)
model.add(ernesto_res, name_prop, name_lit)
model.add(ernesto_res, teaches_prop, inm713_res)
```

Recap: RDF in Java (v)

– Adding triples:

```
model.add(ernesto_res, RDF.type, Person_res)
model.add(ernesto_res, name_prop, name_lit)
model.add(ernesto_res, teaches_prop, inm713_res)
```

– Prefixes: `model.setNsPrefix("city", "http://ex.org/univ/city#");`

SPARQL by Example

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

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/`

SPARQL Examples (i)

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

SPARQL Examples (i)

- DBpedia: RDF version of Wikipedia with 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 DISTINCT ?jd WHERE {
    ?jd foaf:name "Johnny Depp" .
}
```

SPARQL Examples (i)

- DBpedia: RDF version of Wikipedia with 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 DISTINCT ?jd WHERE {
    ?jd foaf:name "Johnny Depp" .
}
```

Answer:

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

SPARQL Examples (ii)

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 .
}
```

SPARQL Examples (ii)

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.)

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

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

Simple Examples (cont.)

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

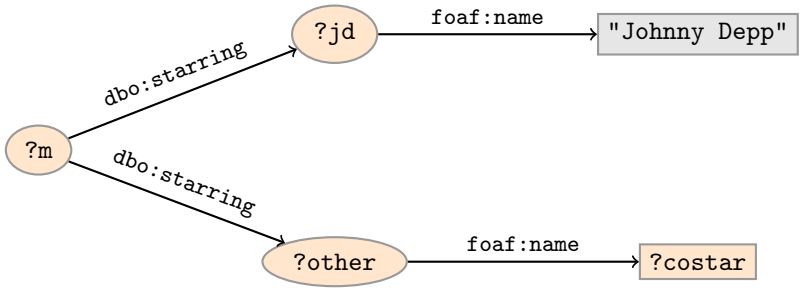
```
SELECT DISTINCT ?costar WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
    ?m dbo:starring ?other .
    ?other foaf:name ?costar .
}
```

Answer:

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

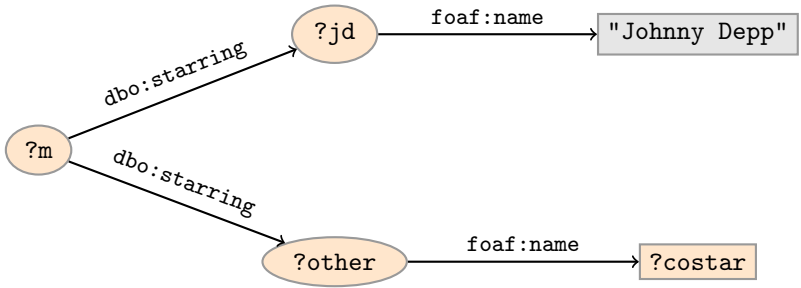
Graph Patterns

The previous SPARQL query as a graph:



Graph Patterns

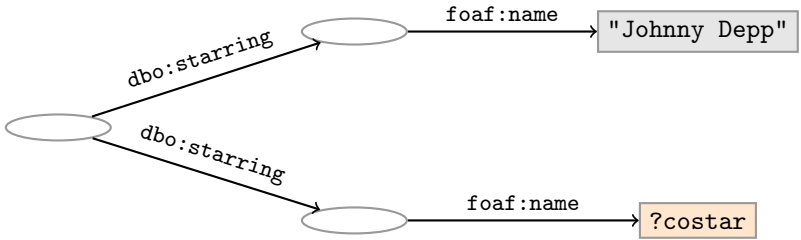
The previous SPARQL query as a graph:



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

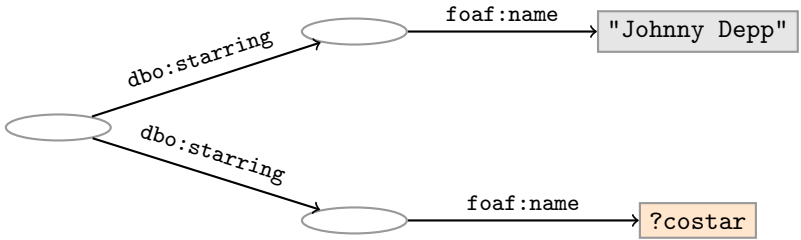
Graph with blank nodes

Variables not SELECTed can equivalently be blank:



Graph with blank nodes

Variables not SELECTed can equivalently be blank:



Pattern matching: a function that assigns values (*i.e.*, resource, a blank node, or a literal) 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 ?costar WHERE {
    _:jd foaf:name "Johnny Depp" .
    _:m dbo:starring _:jd .
    _:m dbo:starring _:other .
    _:other foaf:name ?costar.
}
```

SPARQL Query with blank nodes

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

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

The same with blank node syntax

```
SELECT DISTINCT ?costar WHERE {
    _:m dbo:starring [foaf:name "Johnny Depp"] .
    _:m dbo:starring _:other .
    _:other foaf:name ?costar.
}
```

SPARQL Query with blank nodes

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

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

The same with blank node syntax

```
SELECT DISTINCT ?costar WHERE {
    _:m dbo:starring [foaf:name "Johnny Depp"] .
    _:m dbo:starring [foaf:name ?costar] .
}
```

SPARQL Query with blank nodes

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

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

The same with blank node syntax

```
SELECT DISTINCT ?costar WHERE {
    [ dbo:starring [foaf:name "Johnny Depp"] ;
      dbo:starring [foaf:name ?costar]
    ]
}
```

SPARQL Query with blank nodes

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

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

The same with blank node syntax

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

SPARQL Systematically

Components of an SPARQL query

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

Components of an SPARQL query

Prologue: prefix definitions

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

Components of an SPARQL query

Results: (1) variable list, (2) query type (SELECT, ASK, CONSTRUCT, DESCRIBE), (3) remove duplicates (DISTINCT, REDUCED)

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

Components of an SPARQL query

Dataset specification

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

Components of an SPARQL query

Query pattern: graph pattern to be matched

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

Components of an SPARQL query

Solution modifiers: ORDER BY, LIMIT, OFFSET

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

Types of Queries (i)

SELECT Compute table of bindings for variables

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

Types of Queries (i)

SELECT Compute table of bindings for variables

```
SELECT DISTINCT ?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 (ii)

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

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

Types of Queries (ii)

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

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

DESCRIBE Returns an RDF graph with data about matching resources

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

SPARQL Systematically: Solution Modifiers

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

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). Applied in this order:
  - Order
  - Projection
  - Distinct
  - Reduced
  - Offset
  - Limit

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

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)
```

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.

ORDER BY (Example)

```
SELECT DISTINCT ?costar
FROM <http://dbpedia_dataset>
WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
    ?m dbo:starring ?other .
    ?other foaf:name ?costar .
    FILTER (STR(?costar)!="Johnny Depp")
}
ORDER BY ?costar
```

Projection, DISTINCT, REDUCED

- **Projection** means that only some variables are part of the solution
  - Done with `SELECT ?x ?y WHERE {?x ?y ?z...}`

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 variables** as another 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 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
  - 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

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
- For example, solutions number 51 to 60:

```
SELECT ... WHERE {...} ORDER BY ...
LIMIT 10 OFFSET 50
```

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
- For example, solutions number 51 to 60:

```
SELECT ... WHERE {...} ORDER BY ...
LIMIT 10 OFFSET 50
```

- LIMIT and OFFSET can be used separately
- `OFFSET` not meaningful without `ORDER BY`.

OFFSET and LIMIT (Example)

```
SELECT DISTINCT ?costar
FROM <http://dbpedia_dataset>
WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
    ?m dbo:starring ?other .
    ?other foaf:name ?costar .
    FILTER (STR(?costar)!="Johnny Depp")
}
ORDER BY ?costar
LIMIT 10 OFFSET 50
```

SPARQL Systematically: Query Graph Patterns

Query patterns

- Types of graph patterns for the query pattern (**WHERE clause**):
  - Basic Graph Patterns (BGP)
  - 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.

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

- Scope of blank node labels is the BGP

Basic Graph Patterns (BGP)

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

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

- Scope of blank node labels is the BGP
- **Pattern matching**: a function that maps
  (i) every variable and every blank node in the pattern
  (ii) to a resource, a blank node, or a literal in the RDF graph.

Filters (i)

- A set of triple patterns may include **constraints** or **filters**
- Reduces matches of surrounding group where filter applies
- Example:

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

Filters (ii)

– Example:

```
WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
    ?m dbo:starring ?other .
    ?other foaf:name ?costar .
    FILTER (STR(?costar)!="Johnny Depp")
}
```

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)`, `year(?date)`, `xsd:integer(?value)`
- `regex` is used to match a variable with a regular expression. *Always use with* `str(?var)`. E.g.: `regex(str(?costar), "Alpacino")`.

More details in specification: `http://www.w3.org/TR/rdf-sparql-query/`

OPTIONAL Patterns

– Allows a match to leave some variables **unbound** (e.g. no data is available). *e.g.*,:

```
WHERE {
  ?x a dbo:Person ;
     foaf:name ?name .
  OPTIONAL {
    ?x dbo:birthDate ?date .
  }
}
```

OPTIONAL Patterns

– Allows a match to leave some variables **unbound** (e.g. no data is available). *e.g.*,:

```
WHERE {
  ?x a dbo:Person ;
     foaf:name ?name .
  OPTIONAL {
    ?x dbo:birthDate ?date .
  }
}
```

– `?x` and `?name` bound in every match, `?date` is **bound if available**.

OPTIONAL Patterns

– Allows a match to leave some variables **unbound** (e.g. no data is available). *e.g.*,:

```
WHERE {
  ?x a dbo:Person ;
     foaf:name ?name .
  OPTIONAL {
    ?x dbo:birthDate ?date .
  }
}
```

– `?x` and `?name` bound in every match, `?date` is **bound if available**.

– Groups can contain several **optional parts**, evaluated separately

OPTIONAL Patterns: with FILTER

– Example:

```
WHERE {
  ?x a dbo:Person ;
     foaf:name ?name .
  OPTIONAL {
    ?x dbo:birthDate ?date .
    FILTER (?date > "1980-01-01T00:00:00"^^xsd:dateTime)
  }
}
```

– `?x` and `?name` bound in every match, `?date` is **bound if available** and **from 1980 onwards**.

OPTIONAL Patterns: Negation as Failure

- Testing if a graph pattern is not expressed.
- An OPTIONAL graph pattern introduces the variable.
- FILTER tests the variable is not bound.

OPTIONAL Patterns: Negation as Failure

- Testing if a graph pattern is not expressed.
- An OPTIONAL graph pattern introduces the variable.
- FILTER tests the variable is not bound.
- E.g.

```
WHERE {
  ?x a dbo:Person ;
     foaf:name ?name .
  OPTIONAL {
    ?x dbo:birthDate ?date .
    FILTER (!bound(?date))
  }
}
```

Matching Alternatives (UNION)

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

```
SELECT DISTINCT ?writer
WHERE{
  ?s rdf:type dbo:Book .
  {
    ?s dbo:author ?writer .
  }
  UNION
  {
    ?s dbo:writer ?writer .
  }
}
```

Graph Graph Patterns (RDF datasets)

- SPARQL queries are executed against an **RDF dataset**
- An RDF dataset comprises
  - One **default graph** (unnamed) graph. Target for this week.
  - Zero or more **named graphs** identified by an URI

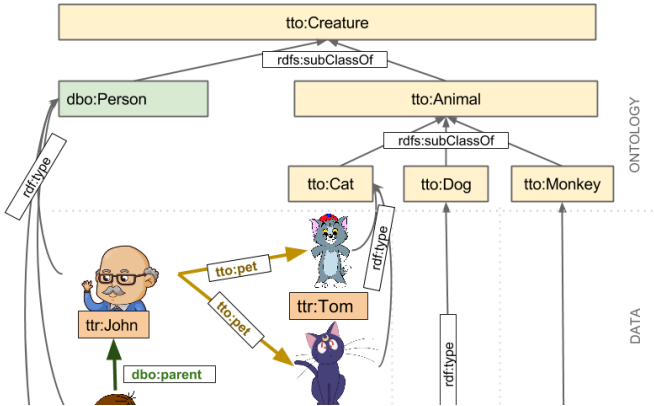
Graph Graph Patterns (RDF datasets)

- SPARQL queries are executed against an **RDF dataset**
- An RDF dataset comprises
  - One **default graph** (unnamed) graph. Target for this week.
  - Zero or more **named graphs** identified by an URI
- FROM and FROM NAMED keywords allows to select an RDF dataset
- Keyword GRAPH makes the named graphs the **active graph** for pattern matching

Laboratory: Hands-on SPARQL

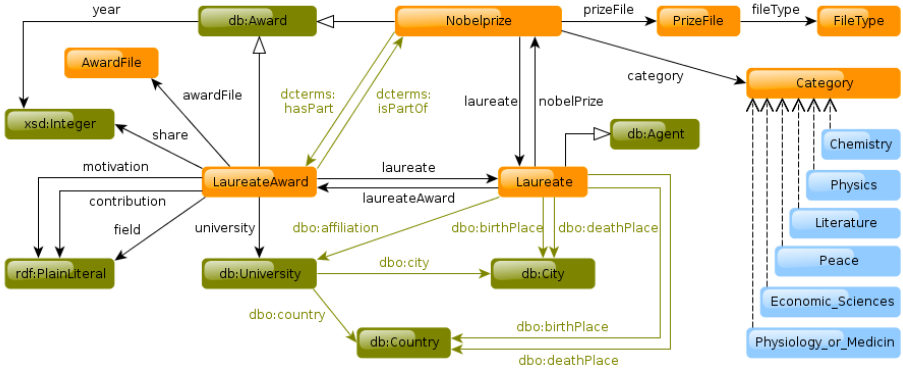
SPARQL Playground

– Platform to learn SPARQL: `http://sparql-playground.sib.swiss/`



Nobel Prize Knowledge Graph

– https://www.nobelprize.org/about/linked-data-examples/



SPARQL in Python: Querying Local Graph with RDFLib

– Querying a local Graph:

```
qres = g.query(
    """SELECT ?thing ?name WHRE {
        ?thing tto:sex "female" .
        ?thing dbp:name ?name .
    }""")
```

– Iterate over the results:

```
for row in qres:
    print("%s is female with name ‘%s’" % (str(row.thing),str(row.name)))
```

– `row` is a dictionary with the RDF terms that match the output variables.

SPARQL in Python: Remote Access with SPARQLWrapper (i)

- SPARQLWrapper: deals with the connection to a SPARQL endpoint
- A SPARQL Endpoint is a service to receive and process SPARQL queries following a protocol.
- Connection: `sparql_web = SPARQLWrapper("http://data.nobelprize.org/sparql")`
- Set results format (default XML): `sparql_web.setReturnFormat(JSON)`

SPARQL in Python: Remote Access with SPARQLWrapper (ii)

– Set SPARQL query:

```
sparql_web.setQuery("""
    SELECT DISTINCT ?name_laur WHERE {
        ?laur rdf:type nobel:Laureate .
        ?laur rdfs:label ?name_laur .
        ?laur foaf:gender "female" .  }
""")
```

– Get (json) results: `results = sparql_web.query().convert()`

– Iterate over the (json) results:

```
for result in results["results"]["bindings"]:
    print(result["name_laur"]["value"])
```

SPARQL in Java: Querying Local Graph with Jena API (i)

– Set query:

```
Query q = QueryFactory.create(
    "PREFIX ttr:  <http://example.org/tuto/resource#>" +
    "PREFIX tto:  <http://example.org/tuto/ontology#>" +
    "PREFIX dbp:  <http://dbpedia.org/property/>" +
    "SELECT ?thing ?name WHERE {" +
        "?thing tto:sex ‘female’ ." +
        "?thing dbp:name ?name ." +
    "}")
```

– Execute query:

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

SPARQL in Java: Querying Local Graph with Jena API (ii)

– Iterate over the query results:

```
while( res.hasNext())
    QuerySolution soln = res.next();
    RDFNode thing = soln.get("?thing");
    RDFNode name = soln.get("?name");
```

– `soln` contain the RDF terms that match the output variables.

SPARQL in Java: Remote Access with Jena API (ii)

- Similar to local graph access.
- Minor query execution change:

```
QueryExecution qe = QueryExecutionFactory
    .sparqlService("http://data.nobelprize.org/sparql",q);
```