SPARQL Query Language

Ernesto Jiménez-Ruiz

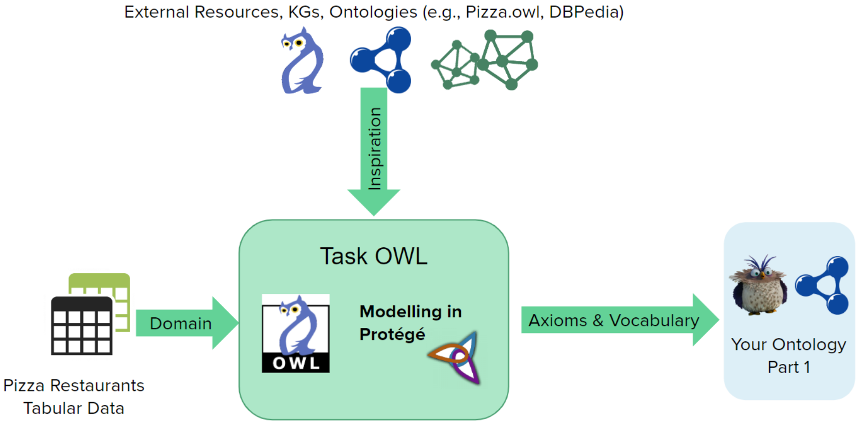
Lecturer in Artificial Intelligence

Before we start...

Coursework Part 1: Ontology modelling (i)

- Part 1 (20%): **creation of an ontology** that covers the knowledge of a given domain. **Deadline**: Sunday, 3 March 2024, 5:00 PM
- **Work in pairs**, or individually. Please register by February 23.

Coursework Part 1: Ontology modelling (ii)

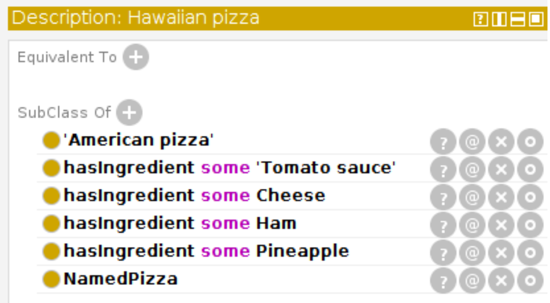

External Resources, KGs, Ontologies (e.g., Pizza.owl, DBPedia)
Inspiration
Task OWL
OWL
Modelling in Protégé
Domain
Pizza Restaurants Tabular Data
Axioms & Vocabulary
Your Ontology Part 1

(*) Ontology CW2 = Model solution ≠ Your Ontology Part 1

OWL 2 Important Examples

Necessary conditions and primitive classes

Hawaiian pizza **implies** having pineapple as ingredient (among others); but not the other way round.



Description: Hawaiian pizza

Equivalent To

SubClass Of

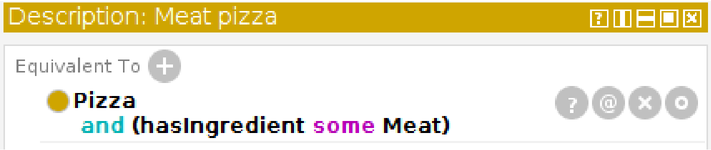
- 'American pizza'
- hasIngredient some 'Tomato sauce'
- hasIngredient some Cheese
- hasIngredient some Ham
- hasIngredient some Pineapple
- NamedPizza

Sufficient conditions and defined classes

Meat pizza **implies** having meat as ingredient (and being pizza).

A pizza with meat as ingredient **implies** being a meat pizza.

Hawaiian pizzas have ham as ingredient and thus they are meat pizzas.



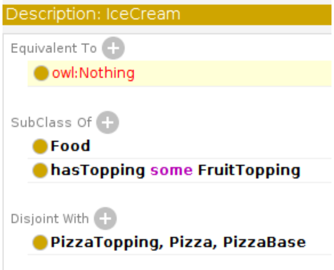
Description: Meat pizza

Equivalent To

Pizza
 and (hasIngredient some Meat)

Detecting modelling errors

- Ice cream **implies** having fruit as topping
- Ice cream is **disjoint with** Pizza
- The domain of has topping is pizza, that is, having any topping **implies** being a pizza.
- Domain is a type of <u>sufficient condition</u>, <u>global scope</u> for the property.

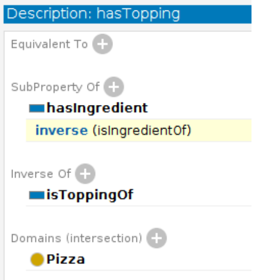


Description: IceCream

Equivalent To
● owl:Nothing

SubClass Of
● Food
● hasTopping some FruitTopping

Disjoint With
● PizzaTopping, Pizza, PizzaBase



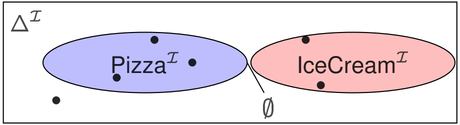
Description: hasTopping

Equivalent To

SubProperty Of
▬ hasIngredient
inverse (isIngredientOf)

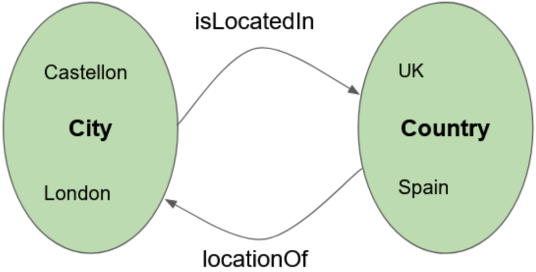
Inverse Of
▬ isToppingOf

Domains (intersection)
● Pizza

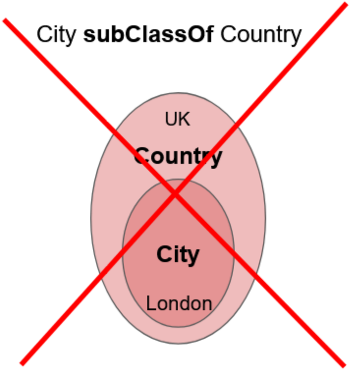


Common mistakes: part-of VS subclass-of

City **subClassOf** isLocatedIn **some** Country

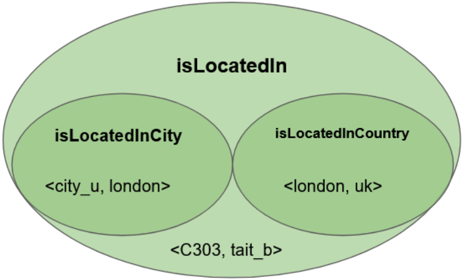


City **subClassOf** Country

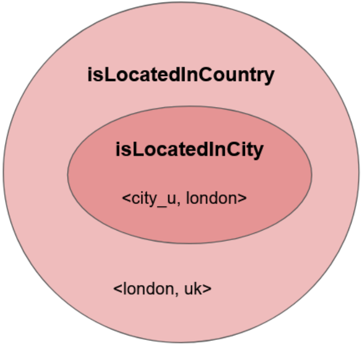


Common mistakes: property hierarchy

isLocatedInCity **subPropertyOf** isLocatedIn
isLocatedInCountry **subPropertyOf** isLocatedIn

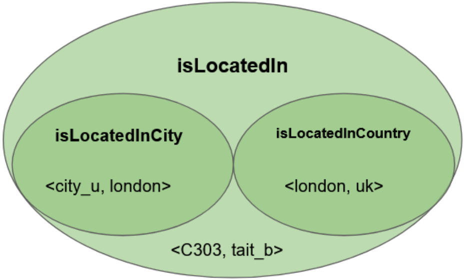


isLocatedInCity **subPropertyOf**
isLocatedinCountry

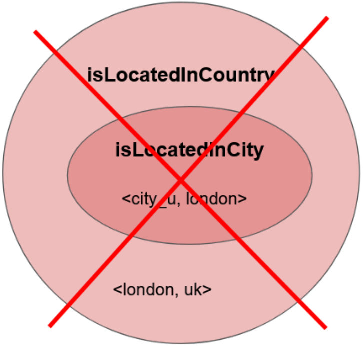


Common mistakes: property hierarchy

isLocatedInCity **subPropertyOf** isLocatedIn
isLocatedInCountry **subPropertyOf** isLocatedIn

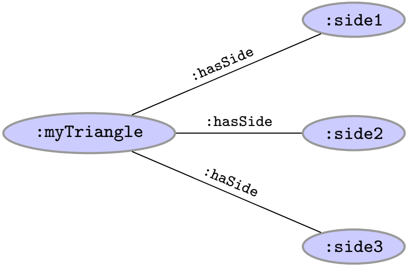


isLocatedInCity **subPropertyOf**
isLocatedinCountry



OWL 2 and Open World Assumption

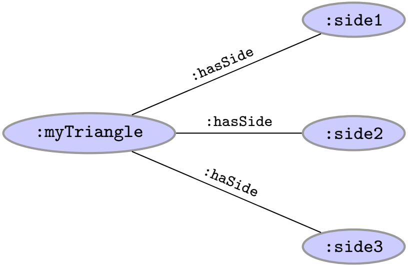
- `:Triangle EquivalentTo :hasSide exactly 3 :Side`



- is `:myTriangle` a `:Triangle`?

OWL 2 and Open World Assumption

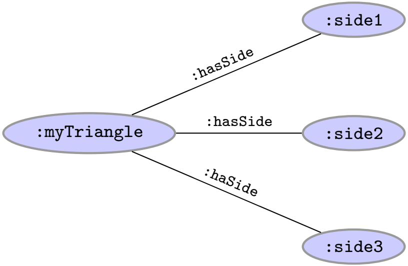
- `:Triangle EquivalentTo :hasSide exactly 3 :Side`



- is `:myTriangle` a `:Triangle`? I don’t know because of OWA and NUNA.

OWL 2 and Open World Assumption

- `:Triangle EquivalentTo :hasSide exactly 3 :Side`



- is `:myTriangle` a `:Triangle`? I don't know because of OWA and NUNA.
- Solution: deductive reasoning complemented with SPARQL queries (in this case with aggregates) → SPARQL 1.1 (not today)

Where are we? Module organization.

✓ Introduction: Becoming a knowledge scientist.

✓ RDF-based knowledge graphs.

✓ OWL ontology language. Focus on modelling.

4. **SPARQL 1.0 Query Language.** (Today)
5. From tabular data to KG.
6. RDFS Semantics and OWL 2 profiles.
7. Ontology Alignment.
8. Ontology (KG) Embeddings and Machine Learning.
9. SPARQL 1.1 and Graph Database solutions.
10. (Large) Language Models and KGs. (Seminar)

Recap: RDF-based Knowledge Graphs

Recap: RDF triples

- RDF talks about *resources* identified by URIs.
- In RDF, all knowledge is represented by *triples* (aka statements or facts)

Recap: RDF triples

- 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 (*e.g.*, `dbr:london rdf:type dbo:City .`)

	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:

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

  - Plain, with language tag:

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

  - Typed, with a URI indicating the type:

    ```
    dbr:london dbo: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:

```
dbr:london rdf:type dbo:City .
dbo:City rdfs:subClassof dbo:PopulatedPlace.
dbr:london rdfs:label "London" .
```

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

Recap: Other vocabularies

Existing vocabularies from KGs like DBpedia:

- Prefix `dbr:` `<http://dbpedia.org/resource/>`
- Prefix `dbo:` `<http://dbpedia.org/ontology/>`
- Prefix `dbp:` `<http://dbpedia.org/property/>`
- Examples:

```
dbr:london rdf:type dbo:City .
```

New vocabularies in the module:

- Prefix `city:` `<http://www.example.org/university/london/city#>`
- Prefix `lab3:` `<http://www.semanticweb.org/ernesto/in3067-inm713/lab3/>`

Recap: RDF Example

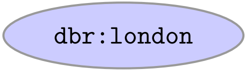
London is a city in England called Londres in Spanish

```
dbr:london a dbo:City .

dbr:london dbo:locationCountry dbr:england .

dbr:london rdfs:label "Londres"@es .
```

Recap: RDF Example

London is a city in England called Londres in Spanish

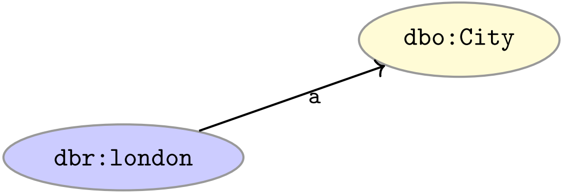
```
dbr:london a dbo:City .
dbr:london dbo:locationCountry dbr:england .
dbr:london rdfs:label "Londres"@es .
```



Recap: RDF Example

London is a city in England called Londres in Spanish

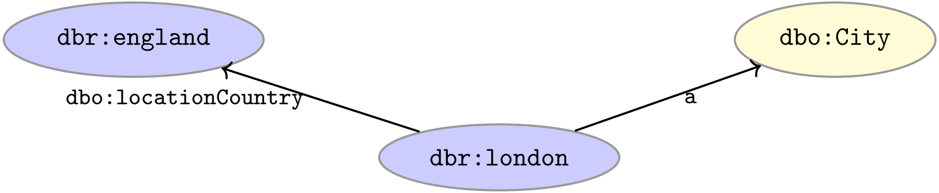
```
dbr:london a dbo:City .
dbr:london dbo:locationCountry dbr:england .
dbr:london rdfs:label "Londres"@es .
```



Recap: RDF Example

London is a city in England called Londres in Spanish

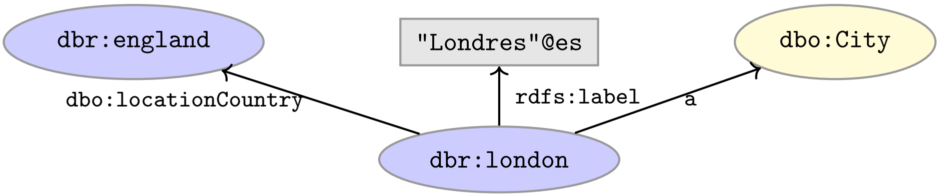
```
dbr:london a dbo:City .
dbr:london dbo:locationCountry dbr:england .
dbr:london rdfs:label "Londres"@es .
```



Recap: RDF Example

London is a city in England called Londres in Spanish

```
dbr:london a dbo:City .
dbr:london dbo:locationCountry dbr:england .
dbr: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 2024 with code IN3067/INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbo:year "2024"^^xsd:gYear .
_:x city:code "IN3067/INM713" .
```

Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2024 with code IN3067/INM713

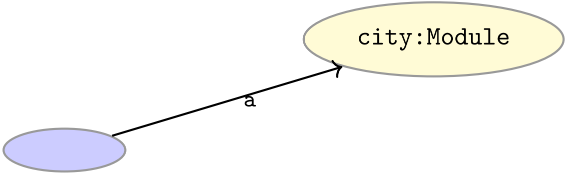
```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbo:year "2024"^^xsd:gYear .
_:x city:code "IN3067/INM713" .
```

Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2024 with code IN3067/INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbo:year "2024"^^xsd:gYear .
_:x city:code "IN3067/INM713" .
```

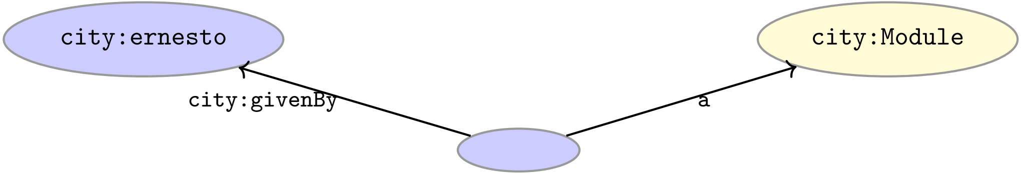


Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2024 with code IN3067/INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbo:year "2024"^^xsd:gYear .
_:x city:code "IN3067/INM713" .
```

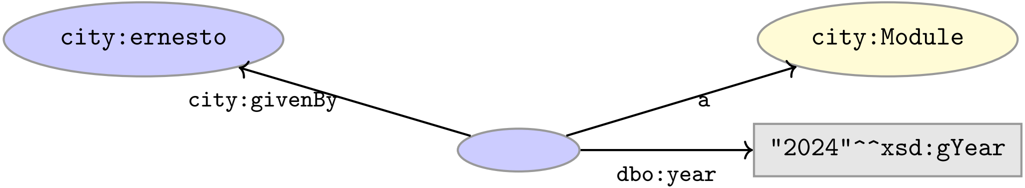


Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2024 with code IN3067/INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbo:year "2024"^^xsd:gYear .
_:x city:code "IN3067/INM713" .
```

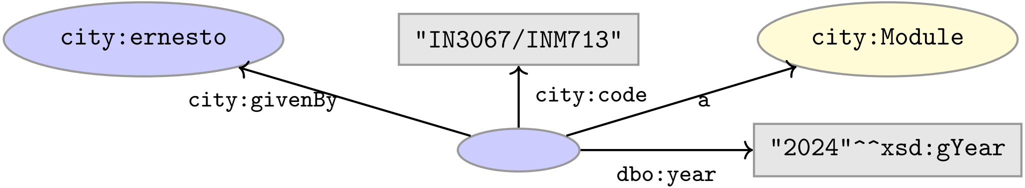


Recap: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a module given by Ernesto in 2024 with code IN3067/INM713

```
_:x a city:Module .
_:x city:givenBy city:ernesto .
_:x dbo:year "2024"^^xsd:gYear .
_:x city:code "IN3067/INM713" .
```



SPARQL by Example

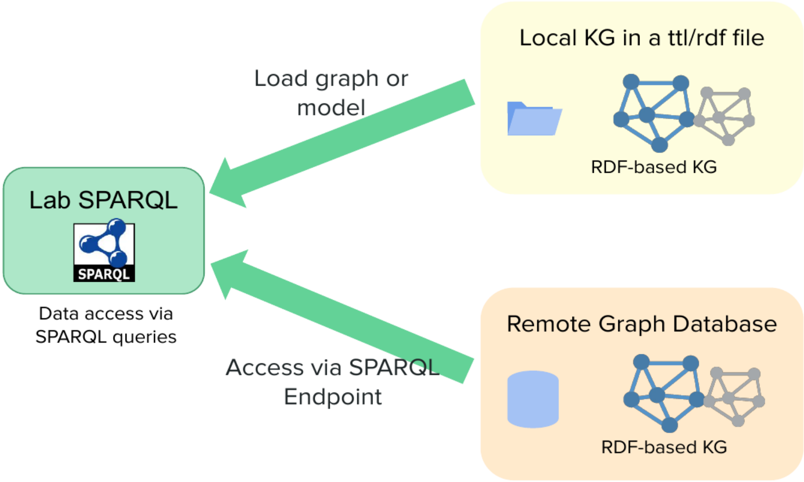
SPARQL

- 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/`
  - Examples: `https://www.w3.org/2008/09/sparql-by-example/`

SPARQL: local and remote KG access


Local KG in a ttl/rdf file
Load graph or model
RDF-based KG
Lab SPARQL
SPARQL
Data access via SPARQL queries
Remote Graph Database
Access via SPARQL Endpoint
RDF-based KG

SPARQL Examples (i)

- Based on 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)

- Based on 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"@en .
}
```

SPARQL Examples (i)

- Based on 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"@en .
}
```

Answer:

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

SPARQL Examples (ii)

Films starring “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 .
}
```

(*) `dbo:starring` comes from the `https://dbpedia.org/ontology/`

SPARQL Examples (ii)

Films starring “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> ...

(*) `dbo:starring` comes from the `https://dbpedia.org/ontology/`

SPARQL Examples (iii)

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 .
}
```

SPARQL Examples (iii)

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

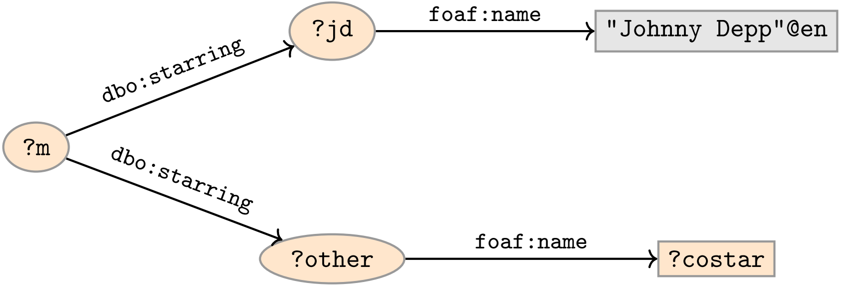
```
SELECT DISTINCT ?costar WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
    ?m dbo:starring ?other .
    ?other foaf:name ?costar .
}
```

Answer:

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

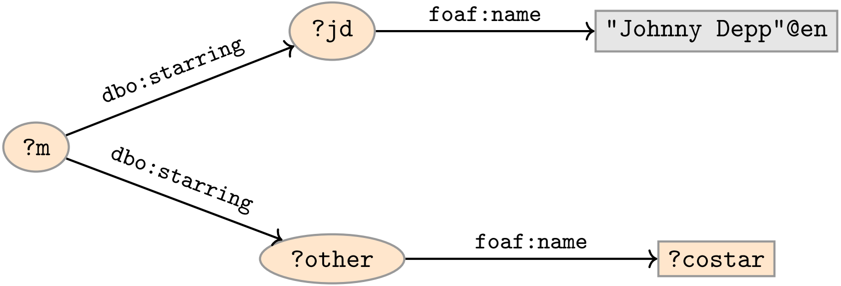
Graph Patterns

The previous SPARQL query as a graph:



Graph Patterns

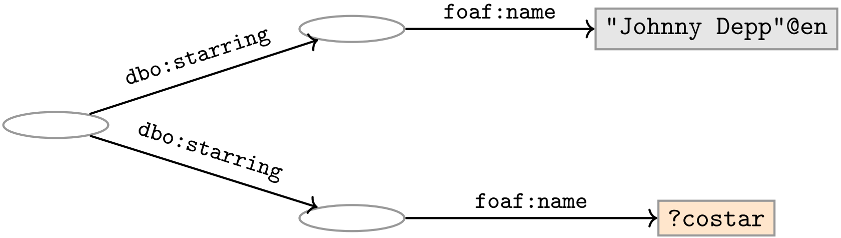
The previous SPARQL query as a graph:



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

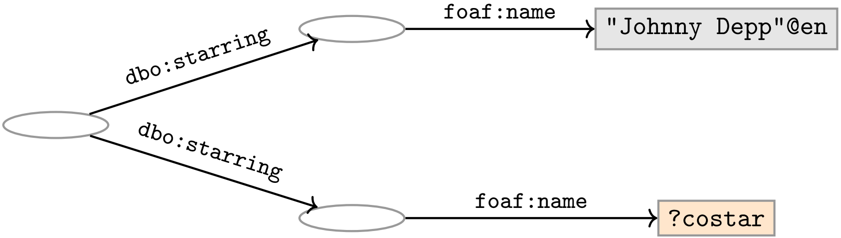
Graph with blank nodes

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"@en .
    _: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"@en .
    _: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"@en] .
    _: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"@en .
    _: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"@en] .
    _: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"@en .
    _: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"@en .
    _: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"@en] ,
                 [foaf:name ?costar]
    ] .
}
```

SPARQL Systematically

Components of a SPARQL query

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX dbo:  <http://dbpedia.org/ontology/>
SELECT DISTINCT ?costar
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
```

Components of a SPARQL query

Prologue: prefix definitions

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX dbo:  <http://dbpedia.org/ontology/>
SELECT DISTINCT ?costar
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
```

Components of a SPARQL query

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

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX dbo:  <http://dbpedia.org/ontology/>
SELECT DISTINCT ?costar
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
```

Components of a 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
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
```

Components of a 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
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
```

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 dbo:country ?country ;
        dbo:populationUrban ?pop .
} ORDER BY ?country 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 dbo:country ?country ;
        dbo:populationUrban ?pop .
} ORDER BY ?country DESC(?pop)
```

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

ORDER BY (Example)

```
SELECT DISTINCT ?costar
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** (*i.e.*, `SELECT`ed variables) means that only some variables are part of the solution
- Done with `SELECT ?x ?y WHERE {?x dbo:starring ?y .  }`

Projection, DISTINCT, REDUCED

- **Projection** (*i.e.*, `SELECT`ed variables) means that only some variables are part of the solution
  - Done with `SELECT ?x ?y WHERE {?x dbo:starring ?y . }`
- DISTINCT **eliminates (all) duplicate** solutions:
  - Done with `SELECT DISTINCT ?x ?y WHERE {?x dbo:starring ?y. }`
  - A solution is a duplicate if it assigns the **same RDF terms to all variables** as another solution.

Projection, DISTINCT, REDUCED

- **Projection** (*i.e.*, `SELECT`ed variables) means that only some variables are part of the solution
  - Done with `SELECT ?x ?y WHERE {?x dbo:starring ?y . }`
- DISTINCT **eliminates (all) duplicate** solutions:
  - Done with `SELECT DISTINCT ?x ?y WHERE {?x dbo:starring ?y. }`
  - 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 dbo:starring ?y . }`
  - Motivation: Can be expensive to find and remove all duplicates
  - Behaviour left to the SPARQL engine.

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 ?x ?y  WHERE {?x dbo:starring ?y .} ORDER BY ?x
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 ?x ?y  WHERE {?x dbo:starring ?y .} ORDER BY ?x
  LIMIT 10 OFFSET 50
  ```

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

OFFSET and LIMIT (Example)

```
SELECT DISTINCT ?costar
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 in between ‘{’ and ‘}’.
- 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 in between ‘{’ and ‘}’.
- 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:

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

Filters (ii)

– 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
```

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. People without a birthdate

```
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)

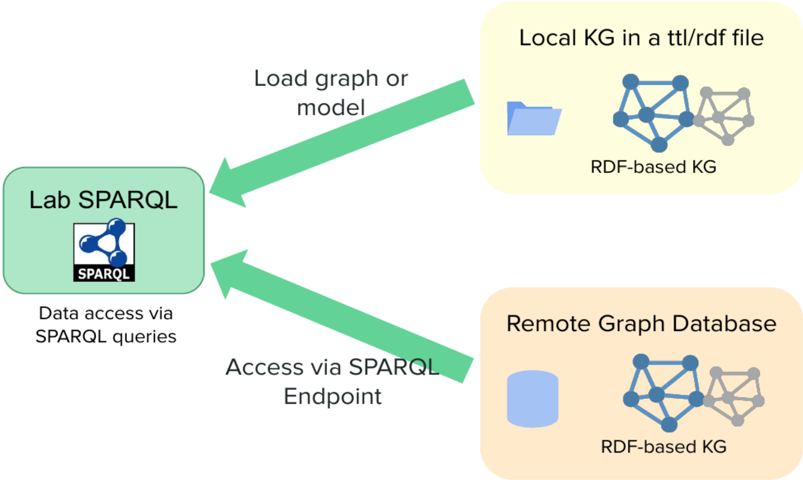
- SPARQL queries are executed against an **RDF dataset**
- An RDF dataset comprises
  - One **default graph** (unnamed) graph. Target for this week.
  - Zero or more **named graphs** identified by an URI

’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
- We will see queries over named graphs in week 10

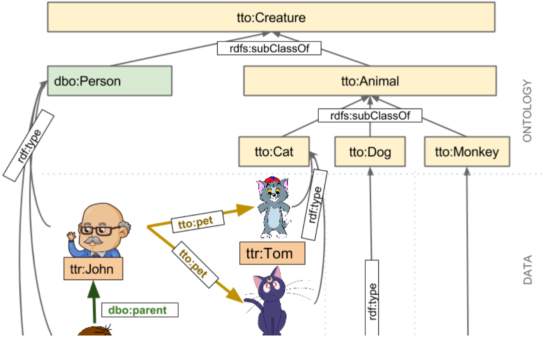
Laboratory: Hands-on SPARQL

SPARQL: local and remote KG access


Local KG in a ttl/rdf file
RDF-based KG
Load graph or model
Lab SPARQL
SPARQL
Data access via SPARQL queries
Access via SPARQL Endpoint
Remote Graph Database
RDF-based KG

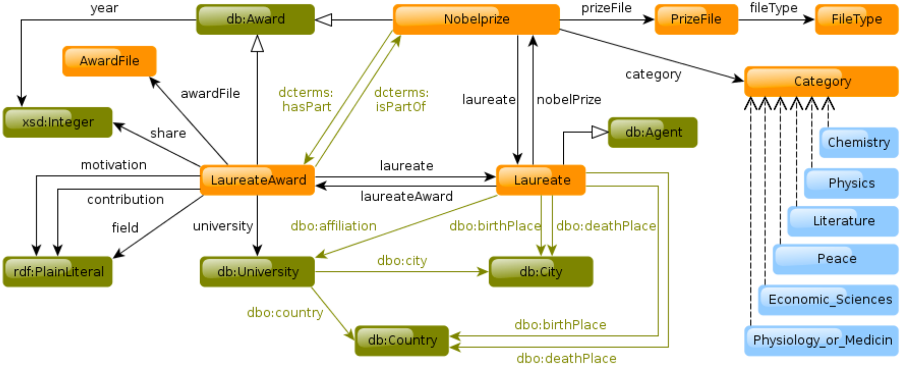
SPARQL Playground

– Based on discontinued platform to learn SPARQL.
  `http://sparql-playground.sib.swiss/`



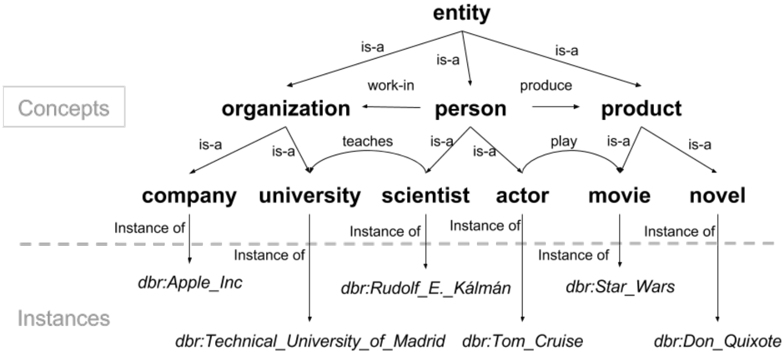
Nobel Prize Knowledge Graph

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



DBpedia Knowledge Graph (i)

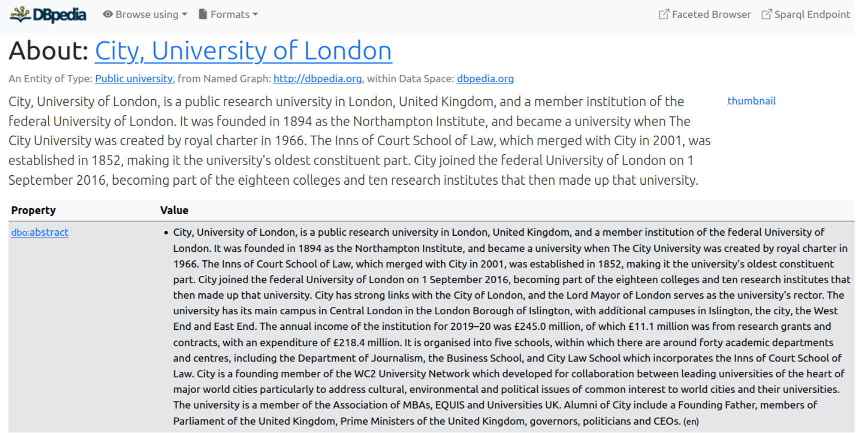
– Ontology/KG: `https://www.dbpedia.org/resources/ontology/`



(*) Image from `https://github.com/gsi-upm/sematch/`

DBpedia Knowledge Graph (ii)

– Linked data Interface: `https://www.dbpedia.org/resources/linked-data/`



About: City, University of London

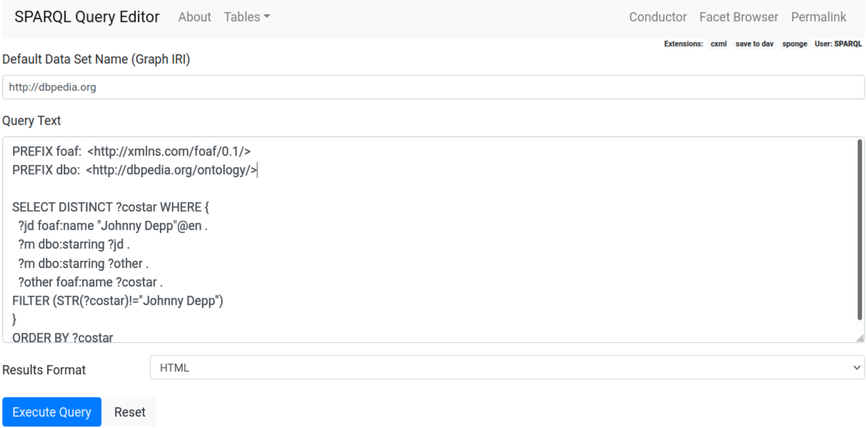
An Entity of Type: Public university, from Named Graph: http://dbpedia.org, within Data Space: dbpedia.org

City, University of London, is a public research university in London, United Kingdom, and a member institution of the federal University of London. It was founded in 1894 as the Northampton Institute, and became a university when The City University was created by royal charter in 1966. The Inns of Court School of Law, which merged with City in 2001, was established in 1852, making it the university's oldest constituent part. City joined the federal University of London on 1 September 2016, becoming part of the eighteen colleges and ten research institutes that then made up that university.

Property	Value
dbo:abstract	• City, University of London, is a public research university in London, United Kingdom, and a member institution of the federal University of London. It was founded in 1894 as the Northampton Institute, and became a university when The City University was created by royal charter in 1966. The Inns of Court School of Law, which merged with City in 2001, was established in 1852, making it the university's oldest constituent part. City joined the federal University of London on 1 September 2016, becoming part of the eighteen colleges and ten research institutes that then made up that university. City has strong links with the City of London, and the Lord Mayor of London serves as the university's rector. The university has its main campus in Central London in the London Borough of Islington, with additional campuses in Islington, the city, the West End and East End. The annual income of the institution for 2019–20 was £245.0 million, of which £11.1 million was from research grants and contracts, with an expenditure of £218.4 million. It is organised into five schools, within which there are around forty academic departments and centres, including the Department of Journalism, the Business School, and City Law School which incorporates the Inns of Court School of Law. City is a founding member of the WC2 University Network which developed for collaboration between leading universities of the heart of major world cities particularly to address cultural, environmental and political issues of common interest to world cities and their universities. The university is a member of the Association of MBAs, EQUIS and Universities UK. Alumni of City include a Founding Father, members of Parliament of the United Kingdom, Prime Ministers of the United Kingdom, governors, politicians and CEOs. (en)

DBPedia Knowledge Graph (iii)

– SPARQL Endpoint: `http://dbpedia.org/sparql`



Default Data Set Name (Graph IRI)

http://dbpedia.org

Query Text

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX dbo:  <http://dbpedia.org/ontology/>

SELECT DISTINCT ?costar 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
```

Results Format: HTML

SPARQL in Java and Python

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://dbpedia.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 ?costar WHERE {
        ?jd foaf:name "Johnny Depp"@en .
        ?m dbo:starring ?jd .
        ?m dbo:starring ?costar .  }
""")
```

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

– Iterate over the (json) results:

```
for result in results["results"]["bindings"]:
    print(result["costar"]["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://dbpedia.org/sparql",q);
```