Reasoning and Querying with Knowledge Graphs

Ernesto Jiménez-Ruiz

Lecturer in Artificial Intelligence

Before we start...

GitHub Repository

`https://github.com/city-knowledge-graphs/phd-course`

Agenda

- Four sessions split into two days
- **Morning sessions**:
  - Theory: 9:00-10:30
  - Break 15 min
  - Hands-on: 10:45-12:15
- Lunch break (1hour)
- **Afternoon sessions**:
  - Theory: 13:15-14:45
  - Break 15 min
  - Hands-on: 15:00-16:30

Course Organization

✓ Introduction to Knowledge Graphs
  ✓ Lab: Creation of a small knowledge graph and ontology.

2. Reasoning and Querying with Knowledge Graphs
   – Lab: First steps with the SPARQL query language.

3. Matching: KG-to-KG and CSV-to-KG
   – Lab: Creation of a (simple) matching system.

4. Knowledge Graphs and Language Models
   – Lab: Ontology Embeddings with OWL2Vec*.

PART I: SPARQL Query Language for RDF-based KGs

SPARQL by Example

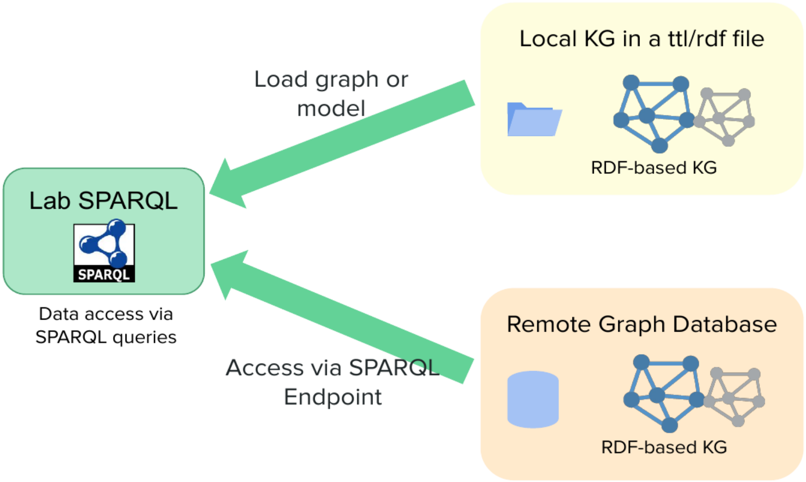
SPARQL

- 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
- Documentation:
  - Syntax and semantics of the SPARQL query language for RDF:
    `http://www.w3.org/TR/rdf-sparql-query/`
    `https://www.w3.org/TR/sparql11-overview/`
  - Examples: `https://www.w3.org/2008/09/sparql-by-example/`

SPARQL: local and remote KG access



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"

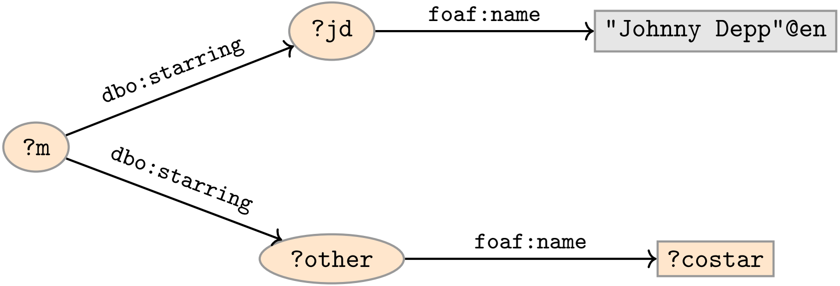
```
SELECT DISTINCT ?costar WHERE {
    ?jd foaf:name "Johnny Depp"@en .
    ?m dbo:starring ?jd .
    ?m dbo:starring ?other .
    ?other foaf:name ?costar .
}
```

Answer:

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

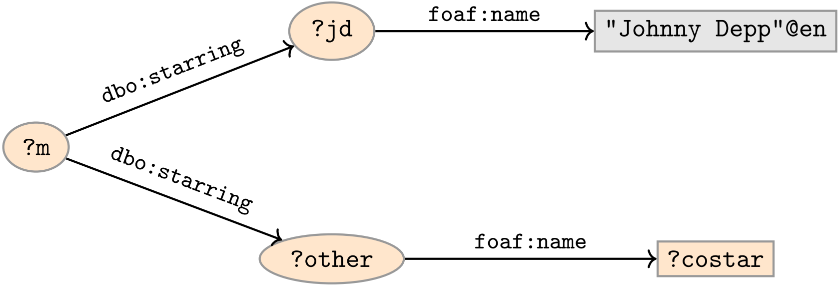
Graph Patterns

The previous SPARQL query as a graph:



Graph Patterns

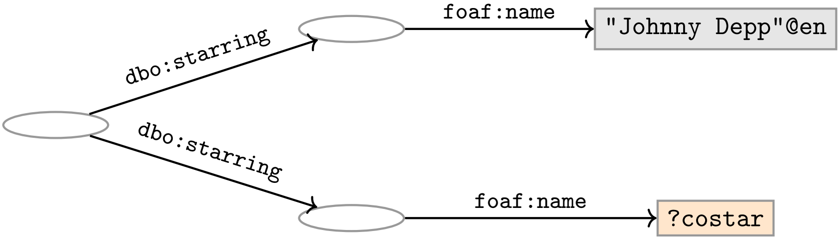
The previous SPARQL query as a graph:



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

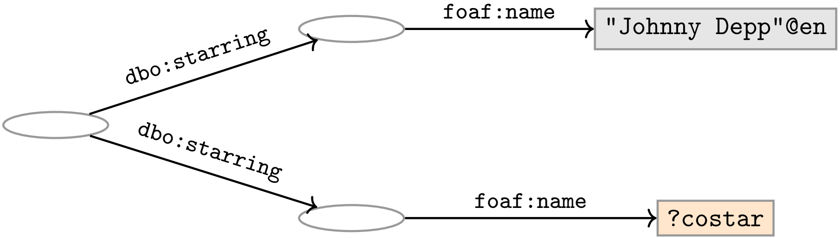
Graph with blank nodes

Variables not `SELECT`ed can equivalently be blank:



Graph with blank nodes

Variables not `SELECT`ed 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 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
- **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)

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**.

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 course.
  - 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 course.
  - 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

SPARQL 1.1

SPARQL 1.1: new features

- The new features in **SPARQL 1.1 QUERY language**:
  - Assignments and Expressions
  - Aggregates
  - Subqueries
  - Negation (new syntax)
  - Property paths

SPARQL 1.1: new features

- The new features in **SPARQL 1.1 QUERY language**:
  - Assignments and Expressions
  - Aggregates
  - Subqueries
  - Negation (new syntax)
  - Property paths
- Specification for:
  - **SPARQL 1.1 UPDATE Language**
  - **SPARQL 1.1 Federated Queries**
  - **SPARQL 1.1 Entailment Regimes**

Assignment and Expressions

- The value of an expression can be assigned/bound to a new variable
- Can be used in SELECT, BIND or GROUP BY clauses:
  (expression AS ?var)

Expressions in SELECT clause

```
SELECT ?city (xsd:integer(?pop)/xsd:float(?area) AS ?density)
{
  ?city dbo:populationTotal ?pop .
  ?city dbo:PopulatedPlace/areaTotal ?area .
  ?city dbo:country dbr:United_Kingdom .
  FILTER (xsd:float(?area)>0.0)
}
```

Aggregates: Grouping and Filtering

- Solutions can optionally be grouped according to one or more expressions.
- To specify the group, use `GROUP BY`.
- If `GROUP BY` is not used, then **only one (implicit) group**

Aggregates: Grouping and Filtering

- Solutions can optionally be grouped according to one or more expressions.
- To specify the group, use `GROUP BY`.
- If `GROUP BY` is not used, then **only one (implicit) group**
- To filter solutions resulting from grouping, use `HAVING`.
- `HAVING` operates over grouped solution sets, in the same way that FILTER operates over un-grouped ones.

Aggregates: Example

Actors with more than 15 movies

```
SELECT ?name (COUNT(?movie) AS ?mcount)
WHERE {
    ?actor foaf:name ?name .
    ?movie dbo:starring ?actor .
}
GROUP BY ?name
HAVING (COUNT(?movie) > 15)
ORDER BY DESC (?mcount)
```

Aggregates: Example

Actors with more than 15 movies

```
SELECT ?name (COUNT(?movie) AS ?mcount)
WHERE {
    ?actor foaf:name ?name .
    ?movie dbo:starring ?actor .
}
GROUP BY ?name
HAVING (COUNT(?movie) > 15)
ORDER BY DESC (?mcount)
```

† Only expressions consisting of aggregates and constants may be projected, together with variables in `GROUP BY`.

Aggregates: common functions

– `Count` counts the number of times a variable has been bound.

– `Sum` sums numerical values of bound variables.

– `Avg` finds the average of numerical values of bound variables.

– `Min` finds the minimum of the numerical values of bound variables.

– `Max` finds the maximum of the numerical values of bound variables.

† Aggregates assume CWA and UNA

Subqueries

- A way to embed SPARQL queries within other queries
- Subqueries are evaluated first and the results are projected to the outer query.

Subqueries

– A way to embed SPARQL queries within other queries
– Subqueries are evaluated first and the results are projected to the outer query.

```
SELECT ?country ?pop (round(?pop/?worldpop*1000)/10 AS ?percentage) WHERE {
  ?country rdf:type dbo:Country .
  ?country dbo:populationTotal ?pop .
  {
    SELECT (sum(?p) AS ?worldpop) WHERE {
      ?c rdf:type dbo:Country .
      ?c dbo:populationTotal ?p .}
  }
}  ORDER BY desc(?pop)
```

Subqueries

- A way to embed SPARQL queries within other queries
- Subqueries are evaluated first and the results are projected to the outer query.

```
SELECT ?country ?pop (round(?pop/?worldpop*1000)/10 AS ?percentage) WHERE {
  ?country rdf:type dbo:Country .
  ?country dbo:populationTotal ?pop .
  {
    SELECT (sum(?p) AS ?worldpop) WHERE {
      ?c rdf:type dbo:Country .
      ?c dbo:populationTotal ?p .}
  }
}  ORDER BY desc(?pop)
```

† Note that the *Sum()* aggregation is done over all the elements (single default group).

Negation in SPARQL 1.1: MINUS and FILTER NOT EXISTS

Two ways to do negation. *e.g.*, retrieve people without a name:

```
SELECT DISTINCT * WHERE {
    ?person a foaf:Person .
    MINUS { ?person foaf:name ?name }
}

SELECT DISTINCT * WHERE {
    ?person a foaf:Person .
    FILTER NOT EXISTS { ?person foaf:name ?name }
}
```

Property paths: basic motivation

- Some queries get needlessly large.
- SPARQL 1.1 define a small language to defined paths.
- Examples:
  - `city:ernesto foaf:knows+ ?friend` to extract all friends of friends.
  - `foaf:maker|dc:creator` instead of `UNION`.
  - Friend's names, `{ _:me foaf:knows/foaf:name ?friendsname }`.
  - Sum several items:

```
SELECT (sum(?cost) AS ?total) { :order :hasItem/:price ?cost }
```

Property paths: example

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

†Similar to blank node syntax.

Property paths: syntax

Syntax Form	Matches
`iri`	An (property) IRI. A path of length one.
`^elt`	Inverse path (object to subject).
`elt1 / elt2`	A sequence path of `elt1` followed by `elt2`.
`elt1 | elt2`	A alternative path of `elt1` or `elt2` (all possibilities are tried).
`elt*`	Seq. of zero or more matches of `elt`.
`elt+`	Seq. of one or more matches of `elt`.
`elt?`	Zero or one matches of `elt`.
`!iri` or `!(`$\mathtt{iri}_1$`| ...|`$\mathtt{iri}_n$`)`	Negated property set.
`!^iri` or `!(^`$\mathtt{iri}_i$`| ...|^`$\mathtt{iri}_n$`)`	Negation of inverse path.
`!(`$\mathtt{iri}_1$`|...|`$\mathtt{iri}_j$`|^`$\mathtt{iri}_{j+1}$`|...|^`$\mathtt{iri}_n$`)`	Negated combination of forward and inverese properties.
`(elt)`	A group path `elt`, brackets control precedence.

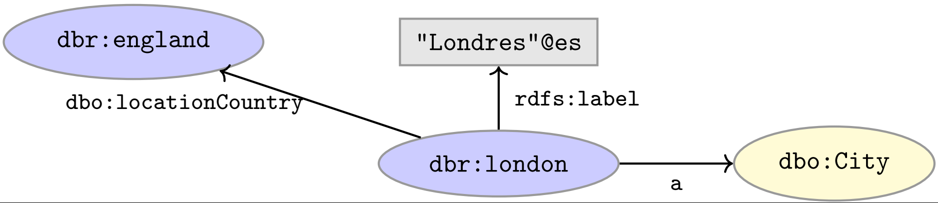
* `elt` is a path element, which may itself be composed of path constructs (see Syntax form).

SPARQL Summary

SPARQL Summary (i)

Return all Cities:

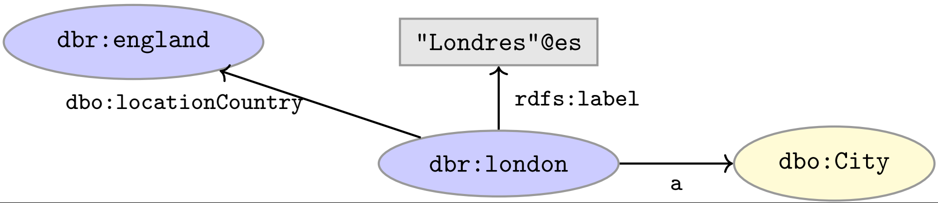
```
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?city WHERE {
    ?city rdf:type dbo:City .
}
```



SPARQL Summary (i)

Return all Cities: Query Result= {dbr:london}

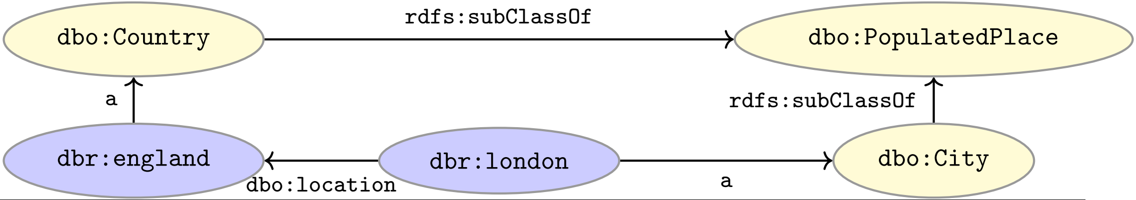
```
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?city WHERE {
    ?city rdf:type dbo:City .
}
```



SPARQL Summary (ii)

Return all Populated Places:

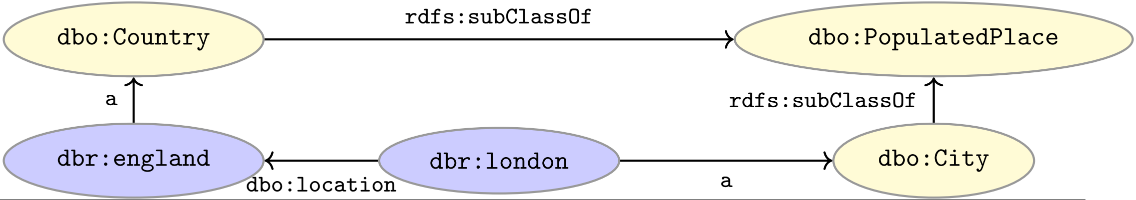
```
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?place WHERE {
    ?place rdf:type dbo:PopulatedPlace .
}
```



SPARQL Summary (ii)

Return all Populated Places: Query Result= {}

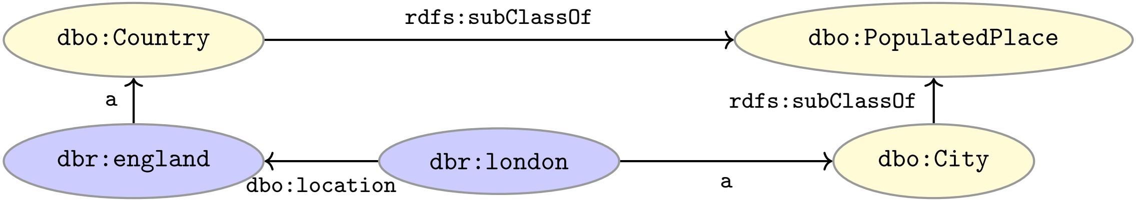
```
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?place WHERE {
    ?place rdf:type dbo:PopulatedPlace .
}
```



PART II: Reasoning with Knowledge Graphs

Implicit Knowledge in KGs: Entailment

- Given a set of triples $\mathcal{G}$ (*i.e.*, a Graph), can we entail a triple t ($\mathcal{G} \models t$)?
- Can we entail the triple: `dbr:london rdf:type dbo:PopulatedPlace` and add it to the graph below? *(Graph expansion via reasoning).*
- Similarly for `dbr:england`



Model-Theoretic Semantics (i)

- **Interpretations** might be conceived as potential "realities" or "worlds".
- Interpretations assign values to elements.
  - *(The **intuitions** behind set-theory are **formally represented**.)*

Model-Theoretic Semantics (i)

- **Interpretations** might be conceived as potential "realities" or "worlds".
- Interpretations assign values to elements.
  - *(The **intuitions** behind set-theory are **formally represented**.)*
- Given an interpretation $\mathcal{I}$ and a set of triples $\mathcal{G}$
- $\mathcal{G}$ is valid in $\mathcal{I}$ (written $\mathcal{I} \models \mathcal{G}$), iff $\mathcal{I} \models t$ for all $t \in \mathcal{G}$.
- Then $\mathcal{I}$ is also called a **model** of $\mathcal{G}$.

Model-Theoretic Semantics (ii)

- The following interpretation $\mathcal{I}$ is a model of our example $\mathcal{G}$:
  - $\texttt{dbo:City}^{\mathcal{I}} = \{\texttt{dbr:london}\}$
  - $\texttt{dbo:Country}^{\mathcal{I}} = \{\texttt{dbr:england}\}$
  - $\texttt{dbo:PopulatedPlace}^{\mathcal{I}} = \{\texttt{dbr:london}, \texttt{dbr:england}\}$
  - $\texttt{dbo:location}^{\mathcal{I}} = \{\langle\texttt{dbr:london, dbr:england}\rangle\}$

Model-Theoretic Semantics (ii)

- The following interpretation $\mathcal{I}$ is a model of our example $\mathcal{G}$:
  - $\texttt{dbo:City}^{\mathcal{I}} = \{\texttt{dbr:london}\}$
  - $\texttt{dbo:Country}^{\mathcal{I}} = \{\texttt{dbr:england}\}$
  - $\texttt{dbo:PopulatedPlace}^{\mathcal{I}} = \{\texttt{dbr:london}, \texttt{dbr:england}\}$
  - $\texttt{dbo:location}^{\mathcal{I}} = \{\langle\texttt{dbr:london, dbr:england}\rangle\}$
- $\mathcal{I} \models \mathcal{G}$ (is a model of $\mathcal{G}$) as the following holds:
  - $\texttt{dbo:City}^{\mathcal{I}} \subseteq \texttt{dbo:PopulatedPlace}^{\mathcal{I}}$
  - $\texttt{dbo:Country}^{\mathcal{I}} \subseteq \texttt{dbo:PopulatedPlace}^{\mathcal{I}}$
  - $\texttt{dbr:london}^{\mathcal{I}} \in \texttt{dbo:City}^{\mathcal{I}}$

Model-Theoretic Semantics (iii)

- t = `dbr:london rdf:type dbo:PopulatedPlace`
- Does $\mathcal{I} \models t$?

Model-Theoretic Semantics (iii)

- t = `dbr:london rdf:type dbo:PopulatedPlace`
- Does $\mathcal{I} \models t$?
  - Yes: $\texttt{dbo:PopulatedPlace}^{\mathcal{I}} = \{\texttt{dbr:london}, \texttt{dbr:england}\}$

Model-Theoretic Semantics (iii)

- t = `dbr:london rdf:type dbo:PopulatedPlace`
- Does $\mathcal{I} \models t$?
  - Yes: $\texttt{dbo:PopulatedPlace}^{\mathcal{I}} = \{\texttt{dbr:london}, \texttt{dbr:england}\}$
- Does $\mathcal{G} \models t$?

Model-Theoretic Semantics (iii)

- t = dbr:london rdf:type dbo:PopulatedPlace
- Does $\mathcal{I} \models t$?
  - Yes: $\texttt{dbo:PopulatedPlace}^{\mathcal{I}} = \{\texttt{dbr:london}, \texttt{dbr:england}\}$
- Does $\mathcal{G} \models t$?
  - **if and only if**
    - For **any interpretation** $\mathcal{I}$ with $\mathcal{I} \models \mathcal{G}$
    - $\mathcal{I} \models t$.
  - (Yes, in this case too.)

Model-Theoretic Semantics (iii)

- t = $\texttt{dbr:london rdf:type dbo:PopulatedPlace}$
- Does $\mathcal{I} \models t$?
  - Yes: $\texttt{dbo:PopulatedPlace}^{\mathcal{I}} = \{\texttt{dbr:london}, \texttt{dbr:england}\}$
- Does $\mathcal{G} \models t$?
  - **if and only if**
    - For **any interpretation** $\mathcal{I}$ with $\mathcal{I} \models \mathcal{G}$
    - $\mathcal{I} \models t$.
  - (Yes, in this case too.)
- Does $\mathcal{G} \models t_2$ (t_2=$\texttt{dbr:london rdf:type dbo:Country}$)?

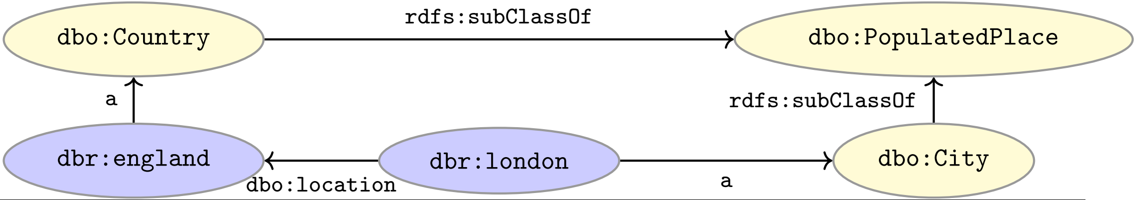
Model-Theoretic Semantics (iii)

- t = `dbr:london rdf:type dbo:PopulatedPlace`
- Does $\mathcal{I} \models t$?
  - Yes: $\texttt{dbo:PopulatedPlace}^{\mathcal{I}} = \{\texttt{dbr:london}, \texttt{dbr:england}\}$
- Does $\mathcal{G} \models t$?
  - **if and only if**
    - For **any interpretation** $\mathcal{I}$ with $\mathcal{I} \models \mathcal{G}$
    - $\mathcal{I} \models t$.
  - (Yes, in this case too.)
- Does $\mathcal{G} \models t_2$ (t_2=`dbr:london rdf:type dbo:Country`)?
  - No: Our $\mathcal{I}$ is a counter example. $\mathcal{I} \models \mathcal{G}$ but $\mathcal{I} \not\models t_2$

SPARQL Example: with entailment

Return all Populated places:

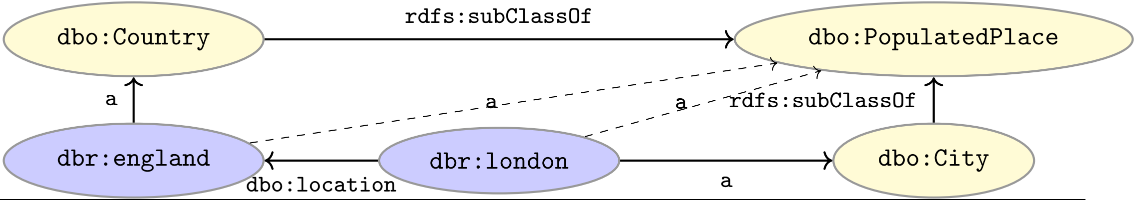
```
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?place WHERE {
    ?place rdf:type dbo:PopulatedPlace .
}
```



SPARQL Example: with entailment

Return all Populated places: **Query Result= {dbr:england, dbr:london}**

```
PREFIX dbo:  <http://dbpedia.org/ontology/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?place WHERE {
    ?place rdf:type dbo:PopulatedPlace .
}
```



Model-Theoretic Semantics in practice

– Model-theoretic semantics yields an unambigous notion of entailment.
– In principle, **all interpretations** need to be considered.
– However there are **infinitely many** such interpretations,
– An **algorithm should terminate** in finite time.

Foundations of Semantic Web Technologies. Chapter 3.

Syntactic methods for Entailment

Syntactic Reasoning

- From the computation point of view, we need means to decide **entailment syntactically**.
- Syntactic methods operate
  - only on the form of a statement, that is on its **concrete grammatical structure** (*i.e.*, triples),
  - without recurring to interpretations.
- Syntactic methods should justify that their so-called **operational semantics** are expected with respect to model-theoretic semantics.

OWL 2 Reasoning Algorithms (i)

– Reasoning in OWL 2 is typically based on **(Hyper)Tableau Reasoning Algorithms** (tableau = truth tree)

– Algorithm tries to construct an **abstraction** of a model.

OWL 2 Reasoning Algorithms (i)

- Reasoning in OWL 2 is typically based on **(Hyper)Tableau Reasoning Algorithms** (tableau = truth tree)
- Algorithm tries to construct an **abstraction** of a model.
- State-of-the-art algorithms:
  - *e.g.*, **HermiT** (default option in Protégé).

OWL 2 Reasoning Algorithms (ii)

- ✓ OWL 2 Reasoners are optimised for TBox tasks.
- ✓ Effective with many realistic ontologies.
- ✗ ABox reasoning tasks are untractable with relatively small KGs.
- ✗ Decidability is open when querying OWL-based KGs with SPARQL.
- – There are some tricks to make it work, but complexity is still high.
- ✗ The main problem is that one cannot just expand the (knowledge) graph and then execute a SPARQL query
- ✓ Solution: **OWL 2 Profiles**

Computational properties: `https://www.w3.org/TR/owl2-profiles/#Computational_Properties`
Seminars by Prof. Ian Horrocks: `http://www.cs.ox.ac.uk/people/ian.horrocks/Seminars/seminars.html`

OWL 1, OWL 2 (profiles) and RDFS

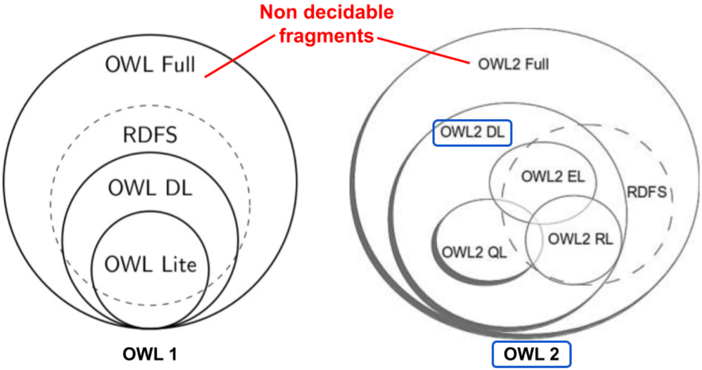
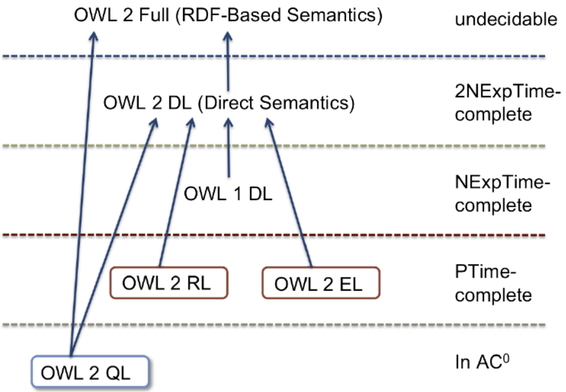


Image adapted from Olivier Cure and Guillaume Blin. RDF Database Systems (Chapter 3). 2015. Elsevier.

Data Complexity in OWL 2 and Profiles



https://www.w3.org/TR/owl2-profiles/

OWL 2 Profiles summary

OWL 2 profiles

- OWL 2 has various **profiles** that correspond to different DLs.
- These profiles have very interesting **computational properties**.

OWL 2 profiles

- OWL 2 has various **profiles** that correspond to different DLs.
- These profiles have very interesting **computational properties**.
  - **OWL 2 QL**:
    - Specifically designed for efficient database integration.

OWL 2 profiles

- OWL 2 has various **profiles** that correspond to different DLs.
- These profiles have very interesting **computational properties**.
  - **OWL 2 QL**:
    - Specifically designed for efficient database integration.
  - **OWL 2 EL**:
    - A lightweight language with polynomial time reasoning.

OWL 2 profiles

- OWL 2 has various **profiles** that correspond to different DLs.
- These profiles have very interesting **computational properties**.
  - **OWL 2 QL**:
    - Specifically designed for efficient database integration.
  - **OWL 2 EL**:
    - A lightweight language with polynomial time reasoning.
  - **OWL 2 RL**:
    - Designed for compatibility with rule-based inference tools.
    - Efficient reasoning with large datasets.

OWL QL Profile

- ✓ Required language so that queries can be rewritten using the TBox.
- ✓ Used in Ontology Based Data Access (OBDA) where SPARQL queries are translated to SQL

Not supported, simplified:

- ✗ disjunction
- ✗ universal quantification, cardinalities, and functional roles
- ✗ $=$ (`SameIndividual`)
- ✗ enumerations (closed classes)
- ✗ subproperties of chains, transitivity
- ✗ reduced list of datatypes

OWL EL Profile

✓ Standard reasoning tasks in P time

✓ Very good for large ontologies.

✓ Used in many biomedical ontologies (*e.g.*, SNOMED CT).

✓ Reasoning can be performed via saturation (*i.e.*, **inference rules**).

Not supported features, simplified:

✗ negation (but $C \sqcap D \sqsubseteq \bot$ possible)

✗ disjunction

✗ universal quantification and cardinalities

✗ inverse roles and some role characteristics

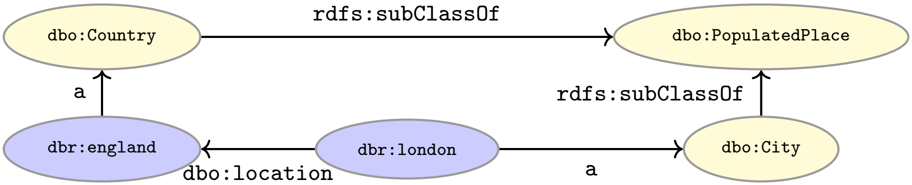
✗ reduced list of datatypes

OWL 2 RL Profile

- ✗ Puts syntactic constraints in the way in which constructs are used (i.e., syntactic subset of OWL 2).
- ✗ Imposes a reduced list of allowed datatypes.
- ✓ OWL 2 RL axioms can be directly translated into **datalog rules**.
- ✓ Enables desirable computational properties using **rule-based** reasoning engines.

Reasoning in the OWL QL Profile

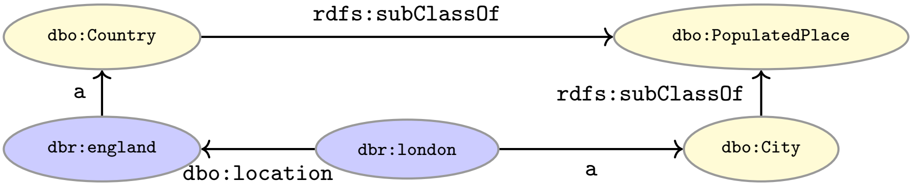
– Reasoning is performed via backward chaining (*e.g.*, rewriting of a given query Q into Q' via the ontology axioms, instead of expanding the graph). For example:



Reasoning in the OWL QL Profile

– Reasoning is performed via backward chaining (*e.g.*, rewriting of a given query Q into Q' via the ontology axioms, instead of expanding the graph). For example:

Q: `SELECT DISTINCT ?place WHERE {?place rdf:type dbo:PopulatedPlace . }`

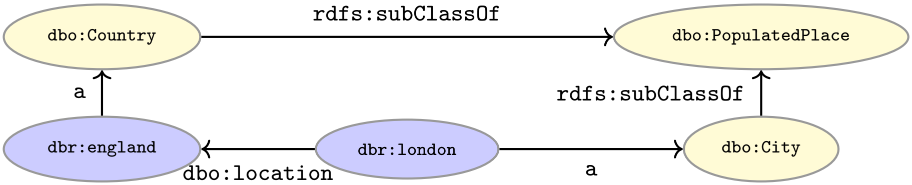


Reasoning in the OWL QL Profile

– Reasoning is performed via backward chaining (*e.g.*, rewriting of a given query Q into Q' via the ontology axioms, instead of expanding the graph). For example:

Q: `SELECT DISTINCT ?place WHERE {?place rdf:type dbo:PopulatedPlace .  }`

Q':
```
SELECT DISTINCT ?place WHERE {
    {?place rdf:type dbo:PopulatedPlace .}
    UNION {?place rdf:type dbo:Country .}
    UNION {?place rdf:type dbo:City .}  }
```



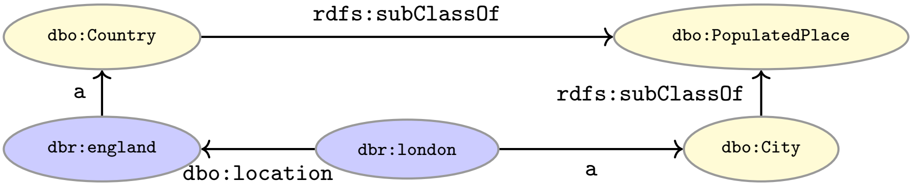
Reasoning in the OWL QL Profile

– Reasoning is performed via backward chaining (*e.g.*, rewriting of a given query Q into Q' via the ontology axioms, instead of expanding the graph). For example:

Q: `SELECT DISTINCT ?place WHERE {?place rdf:type dbo:PopulatedPlace . }`

Q':
```
SELECT DISTINCT ?place WHERE {
    {?place rdf:type dbo:PopulatedPlace .}
    UNION {?place rdf:type dbo:Country .}
    UNION {?place rdf:type dbo:City .}  }
```

Q' Result= {dbr:england, dbr:london}



Inference rules

Inference rules (i)

- Inference rules (also known as deduction rules or derivation rules) is an option to **describe syntactic solutions**.

- The general form of an inference rule is:

$$\frac{P_1, \ldots, P_n}{P}$$

  - the P_i are **premises** (body)
  - and P is the **conclusion** (head).

- An inference rule may have,
  - any number of premises (typically one or two),
  - but only one conclusion.

Inference rules (ii)

- Recall that syllogisms (*i.e.*, inference) can be traced back to Aristotle
- Example:

$$\frac{\text{All human are mortal} \qquad \text{Socrates is a human}}{\text{Therefore, Socrates is mortal}}$$

Inference rules (iii)

- The whole set of inference rules given for a logic is called **deduction calculus**.
- $\vdash$ is the **inference relation**, while $\models$ was the entailment relation using model theoretic semantics.
  - We write $\Gamma \vdash P$ if we can deduce P from the premises Γ.

Inference rules (iii)

- The whole set of inference rules given for a logic is called **deduction calculus**.

- $\vdash$ is the **inference relation**, while $\models$ was the entailment relation using model theoretic semantics.
  - We write $\Gamma \vdash P$ if we can deduce P from the premises Γ.

- **In our example:**
  - the **premises** Γ are a **set of triples** (*i.e.*, a (sub)graph $\mathcal{G}$),
  - the **conclusion** P is a **new triple** t
  - After applying the rules to $\mathcal{G}$ we will get an **expanded graph** $\mathcal{G}'$

RDFS Inference Rules

RDFS supports several rules. Organized into three groups:

1. **Type propagation:**
   – “London is a City, all Cities are populated places, so. . . ”
2. **Property propagation:**
   – “London is the capital of England, anything that is capital of a country is also located in that country, so. . . ”
3. **Domain and range propagation:**
   – “Everything that has a capital is a country, so England is a. . . "
   – “Everything that is a capital is a city, so London is a. . . ”

RDFS Entailment Rules: `https://www.w3.org/TR/rdf-mt/#RDFSRules`

Type propagation

- **Members of superclasses:**

$$\frac{\texttt{A rdfs:subClassOf B .} \qquad \texttt{x rdf:type A .}}{\texttt{x rdf:type B .}}\ \text{rdfs9}$$

(*) rdfs9, rdfs10, rdfs11 are the names of the inference rules in the W3C standard.

A, B are classes; x is an instance.

Type propagation

– **Members of superclasses:**

$$\frac{\texttt{A rdfs:subClassOf B .} \qquad \texttt{x rdf:type A .}}{\texttt{x rdf:type B .}}\ \text{rdfs9}$$

– **Reflexivity of sub-class relation:**

$$\frac{\texttt{A rdf:type rdfs:Class .}}{\texttt{A rdfs:subClassOf A .}}\ \text{rdfs10}$$

(*) rdfs9, rdfs10, rdfs11 are the names of the inference rules in the W3C standard.

A, B are classes; x is an instance.

Type propagation

- **Members of superclasses:**

$$\frac{\texttt{A rdfs:subClassOf B .} \qquad \texttt{x rdf:type A .}}{\texttt{x rdf:type B .}}\ \text{rdfs9}$$

- **Reflexivity of sub-class relation:**

$$\frac{\texttt{A rdf:type rdfs:Class .}}{\texttt{A rdfs:subClassOf A .}}\ \text{rdfs10}$$

- **Transitivity of sub-class relation:**

$$\frac{\texttt{A rdfs:subClassOf B .} \qquad \texttt{B rdfs:subClassOf C .}}{\texttt{A rdfs:subClassOf C .}}\ \text{rdfs11}$$

(*) rdfs9, rdfs10, rdfs11 are the names of the inference rules in the W3C standard.

A, B are classes; x is an instance.

Type propagation: Examples

– **Members of superclasses:**

$$\frac{\texttt{:City rdfs:subClassOf :PopulatedPlace .} \qquad \texttt{:london rdf:type :City .}}{\texttt{:london rdf:type :PopulatedPlace .}}\ \text{rdfs9}$$

– **Reflexivity of sub-class relation:**

$$\frac{\texttt{:City rdf:type rdfs:Class .}}{\texttt{:City rdfs:subClassOf :City .}}\ \text{rdfs10}$$

– **Transitivity of sub-class relation:**

$$\frac{\texttt{:City rdfs:subClassOf :PopulatedPlace .} \qquad \texttt{:PopulatedPlace rdfs:subClassOf :Place .}}{\texttt{:City rdfs:subClassOf :Place .}}\ \text{rdfs11}$$

Property Propagation

– **Transitivity:**

$$\frac{\texttt{P rdfs:subPropertyOf Q .} \qquad \texttt{Q rdfs:subPropertyOf R .}}{\texttt{P rdfs:subPropertyOf R .}}\ \text{rdfs5}$$

(*) P, Q are properties; u, v are instances.

Property Propagation

– **Transitivity:**

$$\frac{\texttt{P rdfs:subPropertyOf Q .} \qquad \texttt{Q rdfs:subPropertyOf R .}}{\texttt{P rdfs:subPropertyOf R .}}\ \text{rdfs5}$$

– **Reflexivity:**

$$\frac{\texttt{P rdf:type rdf:Property .}}{\texttt{P rdfs:subPropertyOf P .}}\ \text{rdfs6}$$

(*) P, Q are properties; u, v are instances.

Property Propagation

– **Transitivity:**

$$\frac{\texttt{P rdfs:subPropertyOf Q .} \qquad \texttt{Q rdfs:subPropertyOf R .}}{\texttt{P rdfs:subPropertyOf R .}}\ \text{rdfs5}$$

– **Reflexivity:**

$$\frac{\texttt{P rdf:type rdf:Property .}}{\texttt{P rdfs:subPropertyOf P .}}\ \text{rdfs6}$$

– **Property transfer:**

$$\frac{\texttt{P rdfs:subPropertyOf Q .} \qquad \texttt{u P v .}}{\texttt{u Q v .}}\ \text{rdfs7}$$

(*) P, Q are properties; u, v are instances.

Property Propagation: Examples

– **Transitivity:**

$$\frac{\texttt{:has\_writer rdfs:subPropertyOf :has\_author .} \qquad \texttt{:has\_author rdfs:subPropertyOf :has\_creator .}}{\texttt{:has\_writer rdfs:subPropertyOf :has\_creator .}}\ \text{rdfs5}$$

– **Reflexivity:**

$$\frac{\texttt{:has\_writer rdf:type rdf:Property .}}{\texttt{:has\_writer rdfs:subPropertyOf :has\_writer .}}\ \text{rdfs6}$$

– **Property transfer:**

$$\frac{\texttt{:has\_author rdfs:subPropertyOf :has\_creator .} \qquad \texttt{:Hamlet :has\_author :Shakespeare .}}{\texttt{:Hamlet :has\_creator :Shakespeare .}}\ \text{rdfs7}$$

Domain and range propagation

Typing triggered by the use of properties.

- **Domain propagation:**

$$\frac{\texttt{P rdfs:domain A .} \qquad \texttt{x P y .}}{\texttt{x rdf:type A .}}\ \text{rdfs2}$$

(*) P, Q are properties; x, y are instances.

Domain and range propagation

Typing triggered by the use of properties.

- **Domain propagation:**

$$\frac{\texttt{P rdfs:domain A .} \qquad \texttt{x P y .}}{\texttt{x rdf:type A .}}\ \text{rdfs2}$$

- **Range propagation:**

$$\frac{\texttt{P rdfs:range B .} \qquad \texttt{x P y .}}{\texttt{y rdf:type B .}}\ \text{rdfs3}$$

(*) P, Q are properties; x, y are instances.

Domain and Range Propagation: Examples

– **Domain propagation:**

$$\frac{\texttt{:capitalOf rdfs:domain :City .} \qquad \texttt{:london :capitalOf :england .}}{\texttt{:london rdf:type :City .}}\ \text{rdfs2}$$

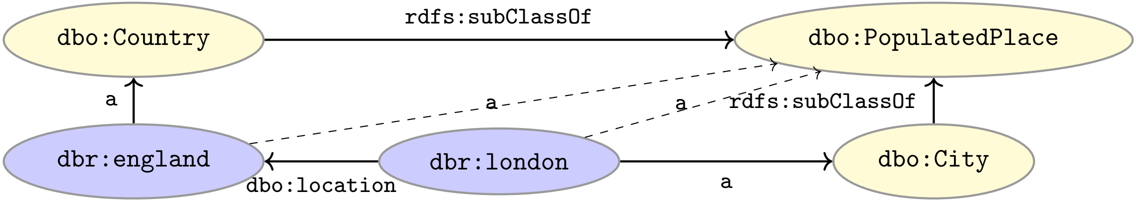
– **Range propagation:**

$$\frac{\texttt{:capitalOf rdfs:range :Country .} \qquad \texttt{:london :capitalOf :england .}}{\texttt{:england rdf:type :Country .}}\ \text{rdfs3}$$

RDFS inference rules in our example

$$\frac{\texttt{dbo:City rdfs:subClassOf dbo:PopulatedPlace .} \qquad \texttt{dbr:london rdf:type dbo:City .}}{\texttt{dbr:london rdf:type dbo:PopulatedPace .}}\ \text{rdfs9}$$

$$\frac{\texttt{dbo:Country rdfs:subClassOf dbo:PopulatedPlace .} \qquad \texttt{dbr:england rdf:type dbo:Country .}}{\texttt{dbr:england rdf:type dbo:PopulatedPlace .}}\ \text{rdfs}$$



Inference rules in the OWL EL Profile

- Reasoning can be performed via saturation† (*i.e.*, inference rules).
- For example:

$$\frac{A \sqsubseteq B \qquad B \sqsubseteq C}{A \sqsubseteq C}$$

$$\frac{A \sqsubseteq \exists R.B \qquad \exists S.B \sqsubseteq C \qquad S \sqsubseteq R}{A \sqsubseteq C}$$

† Using a saturation-based approach over an OWL 2 ontology is not possible.

ELK reasoner (also available as Protégé plugin): `https://github.com/liveontologies/elk-reasoner/wiki`

Inference rules in the OWL 2 RL Profile

– Reasoning via full materialisation of the graph, similarly to RDFS inference rules. *e.g.*,:

$$\frac{\texttt{p1 owl:inverseOf p2 .} \qquad \texttt{?x ?p1 ?y .}}{\texttt{?y ?p2 ?x .}}$$

– See W3C specification for further inference rules in OWL 2 RL in addition to the RDFS ones.

W3C: `https://www.w3.org/TR/owl2-profiles/#Reasoning_in_OWL_2_RL_and_RDF_Graphs_using_Rules`
GraphDB: `https://graphdb.ontotext.com/documentation/standard/reasoning.html`

OWL 2 Important Practical Examples

Open World Assumptions

Closed World Assumption (**CWA**)

- Complete knowledge.
- Any statement that is not known to be true is false. ($*$)
- Typical semantics for **database systems**.

Open World Assumptions

Closed World Assumption (**CWA**)

- Complete knowledge.
- Any statement that is not known to be true is false. ($*$)
- Typical semantics for **database systems**.

Open World Assumption (**OWA**)

- Potential incomplete knowledge.
- ($*$) does not hold.
- Typical semantics for **logic-based systems** (including OWL).

Name Assumptions

Unique Name Assumption (**UNA**)

- Different names **always** denote different things.
  - E.g., $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.
  - common in relational databases.

Name Assumptions

Unique Name Assumption (**UNA**)

- Different names **always** denote different things.
  - E.g., $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.
  - common in relational databases.

Non-unique Name Assumption (**NUNA**)

- Different names **need not** denote different things. **As in OWL**.
  - $\mathtt{dbpedia{:}Person}^{\mathcal{I}} = \mathtt{foaf{:}Person}^{\mathcal{I}}$.
  - $\mathtt{wikidata{:}ernesto}^{\mathcal{I}}$ = $\mathtt{city{:}ernesto}^{\mathcal{I}}$

Name Assumptions

Unique Name Assumption (**UNA**)

- Different names **always** denote different things.
  - E.g., $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.
  - common in relational databases.

Non-unique Name Assumption (**NUNA**)

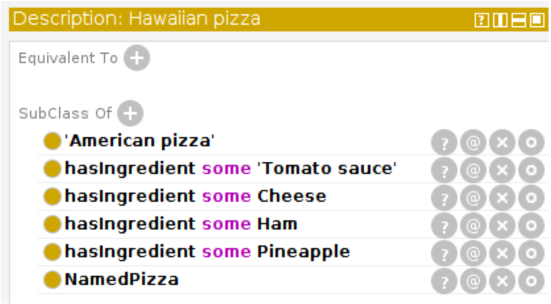
- Different names **need not** denote different things. **As in OWL**.
  - $\texttt{dbpedia:Person}^{\mathcal{I}} = \texttt{foaf:Person}^{\mathcal{I}}$.
  - $\texttt{wikidata:ernesto}^{\mathcal{I}}$ = $\texttt{city:ernesto}^{\mathcal{I}}$

Equal names (*e.g.*, URIs) always denote the same “thing”.

- E.g., cannot have $\texttt{city:ernesto}^{\mathcal{I}} \neq \texttt{city:ernesto}^{\mathcal{I}}$.

Necessary conditions and primitive classes

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

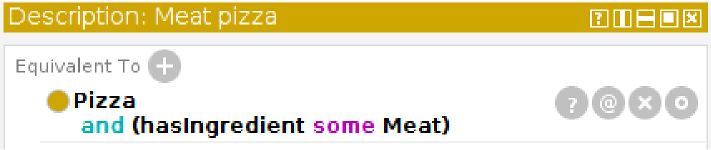


Sufficient conditions and defined classes

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

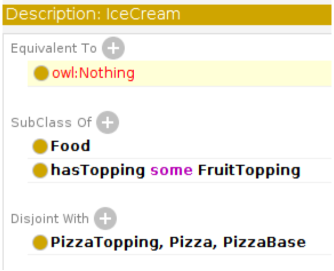
A pizza with meat as ingredient **implies** being a meat pizza.

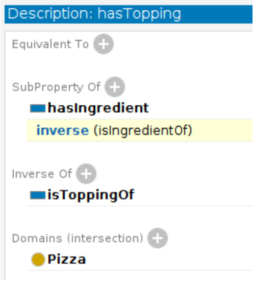
Hawaiian pizzas have ham as ingredient and thus they are meat pizzas.

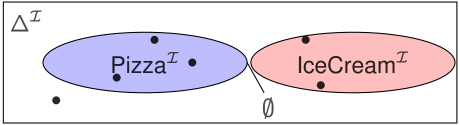


Detecting modelling errors

– 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 sufficient condition, global scope for the property.

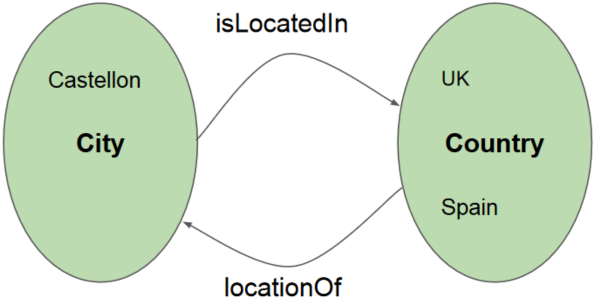




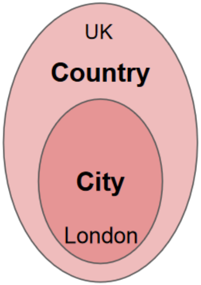


Common mistakes: part-of VS subclass-of (i)

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

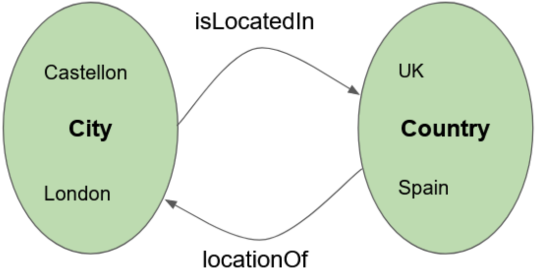


City **subClassOf** Country

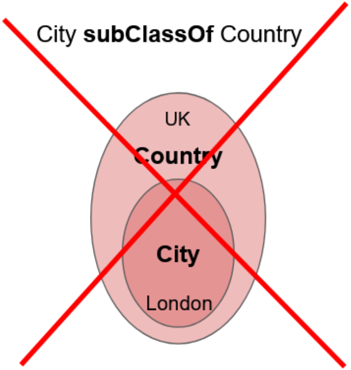


Common mistakes: part-of VS subclass-of (i)

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

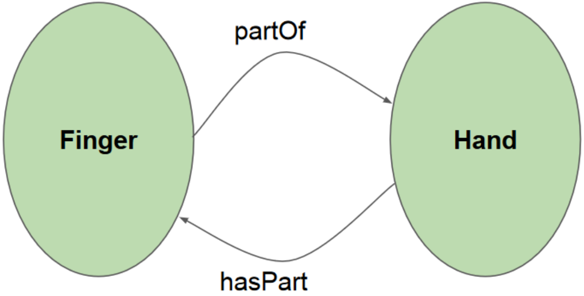


City **subClassOf** Country

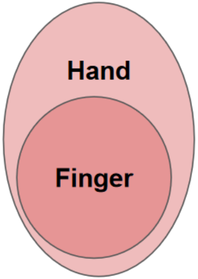


Common mistakes: part-of VS subclass-of (ii)

Finger **subClassOf** partOf **some** Hand

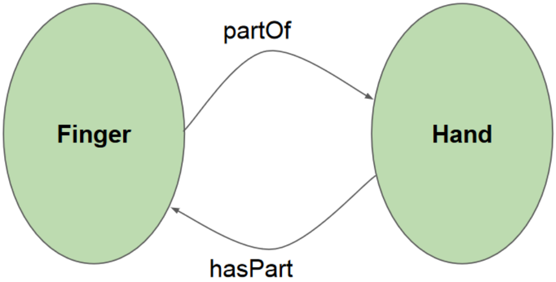


Finger **subClassOf** Hand

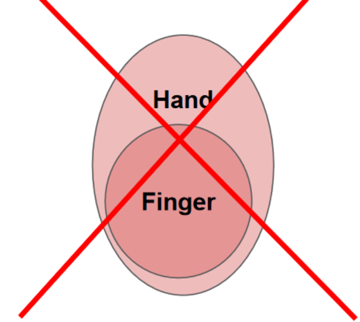


Common mistakes: part-of VS subclass-of (ii)

Finger **subClassOf** partOf **some** Hand

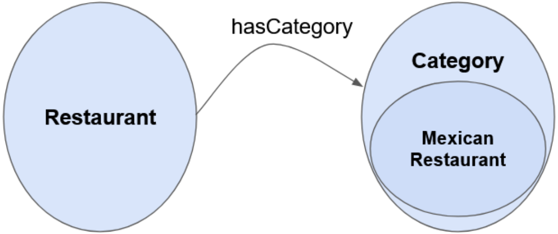


Finger **subClassOf** Hand

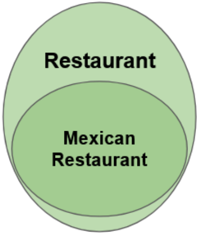


Common mistakes: part-of VS subclass-of (iii)

Restaurant **subClassOf** hasCategory **some** Category

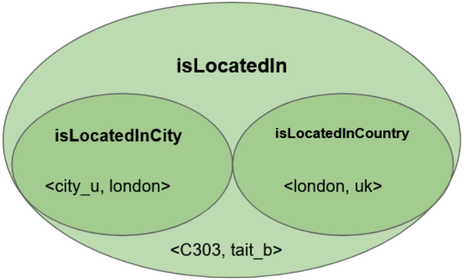


Mexican_Restaurant **subClassOf** Restaurant

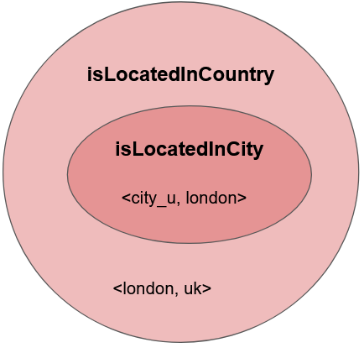


Common mistakes: property hierarchy

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

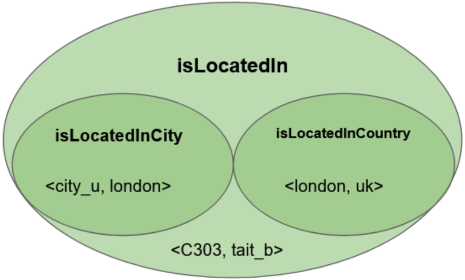


isLocatedInCity **subPropertyOf** isLocatedinCountry

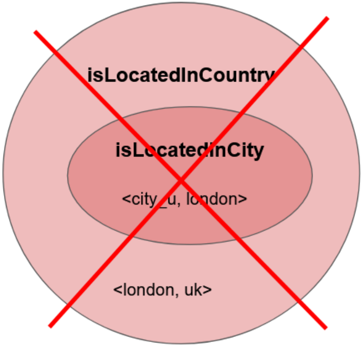


Common mistakes: property hierarchy

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

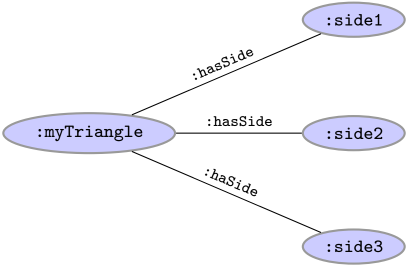


isLocatedInCity **subPropertyOf**
isLocatedinCountry



OWL 2 and Open World Assumption

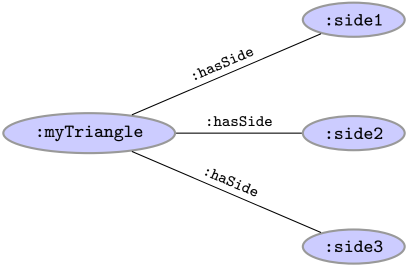
- `:Triangle EquivalentTo :hasSide exactly 3 :Side`



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

OWL 2 and Open World Assumption

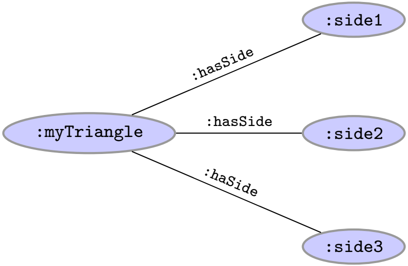
– `:Triangle EquivalentTo :hasSide exactly 3 :Side`



– 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) $\rightarrow$ SPARQL assumes a CWA.

Graph Database Solutions

How to store Graph models?

- Relational model
  - Single table with 3 columns (source, edge, target)
  - Property tables (one table per type of entity, *e.g.*, Person, Module)
  - Binary tables (one table per relationship, *e.g.*, source, target)

M. Wylot and others. RDF Data Storage and Query Processing Schemes. ACM Computing Surveys 2018

How to store Graph models?

- Relational model
  - Single table with 3 columns (source, edge, target)
  - Property tables (one table per type of entity, *e.g.*, Person, Module)
  - Binary tables (one table per relationship, *e.g.*, source, target)

- **Graph-based databases** (NoSQL)

M. Wylot and others. RDF Data Storage and Query Processing Schemes. ACM Computing Surveys 2018

Graph database solutions

- Scale to large Knowledge Graphs.
- Sophisticated indexing structures.
- Optimised reasoning.
- Fast query performance.
- Server solution in production.

State-of-the-art solutions

Dimensions:

- Free version.
- Compliance with Semantic Web standards.
- Reasoning capabilities.
- In-memory or In-disk.
- Documentation and installation requirements.
- Additional features.

A Survey of RDF Stores & SPARQL Engines for Querying Knowledge Graphs. arXiv:2102.13027 2021 (Appendix A)

Semantic Web standards

- The **World Wide Web Consortium (W3C)** is an international community that develops open standards to ensure the long-term growth of the Web: `https://www.w3.org/`
- On the Web and **beyond**.
- **Why standards?**
  - broader industry (and academic) agreement,
  - interoperability across organizations and applications,
  - avoids vendor lock-in of a particular (exchange or query) format.

Apache Jena TDB

– Free solution.

– Provides a native (in-disk) RDF store.

– In combination with Jena Fuseki to provide SPARQL Endpoint support.

✗ Supports reasoning as in Jena, but not direct support for OWL 2 nor the OWL 2 profiles.

`https://jena.apache.org/documentation/tdb/`

OpenLink Virtuoso

✓ Provides the SPARQL endpoint for DBpedia.

– Open source and commercial versions.

– Object-oriented database model.

✓ Native graph model storage provider for Jena and RDF4J.

✗ Custom inference rules. Partial support for OWL 2.

`https://virtuoso.openlinksw.com/`

`http://vos.openlinksw.com/owiki/wiki/VOS`

Blazegraph

- ✓ Provides the SPARQL Endpoint for Wikidata.
- ✓ Free and open source.
- – Both in-memory and disk-oriented storage.
- ✗ Only supports OWL 1 Lite reasoning.
- – Blazegraph team now working for Amazon.

`https://blazegraph.com/`

AllegroGraph

– Free and commercial licenses.

✓ Support for OWL 2 RL materialization.

✓ Client interface in several languages.

– Can be used to query both documents and graph data (via SPARQL).

`https://allegrograph.com/`

Neo4j

- ✓ Open source graph database.
- – Based on the Property Graph Model.
- ✗ Cypher as graph query language (no native SPARQL support).
- – Support *via a plugin* for RDF, RDFS and OWL vocabularies.
- ✗ Basic inferencing support.
- ✓ Support for Analytics.
- ✓ Interfaces in many languages.

`https://neo4j.com/`

GraphDB (formerly OWLIM)

– Free and commercial versions.

✓ Very easy to install and use.

✓ Powerful reasoning features: including OWL 2 QL and RL profiles.

✓ Supports SHACL validation.

– Includes text indexing via lucene.

– Powered the early Linked Data services at the BBC.

`https://www.ontotext.com/products/graphdb/`

RDFox

– Commercial system. Free academic license on request.

✓ Support for materialization-based datalog reasoning (including OWL 2 RL and SWRL rules).

✓ Supports SHACL validation.

– In-memory RDF engine.

✓ Access via Java API or remotely via REST API or SPARQL Endpoint.

✗ Limitation on the size of the memory.

`https://www.oxfordsemantic.tech/product`

Amazon Neptune

– Cloud-based only solution.

– Blazegraph is now part of Amazon Neptune.

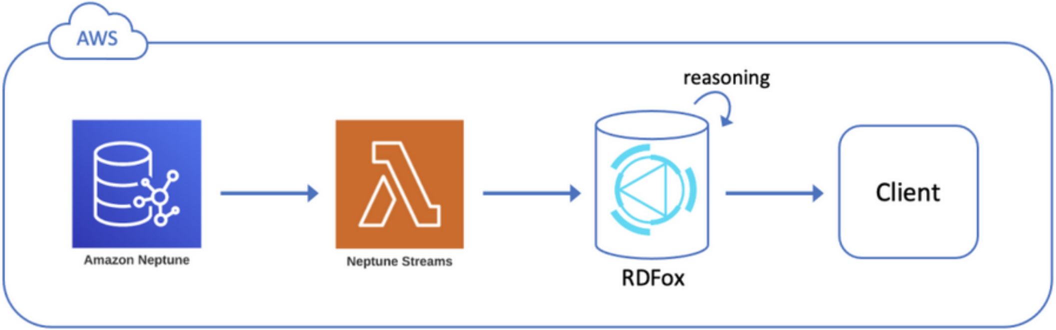
✗ On-Demand pricing.

✗ Native inferencing is not yet supported.

✓ Keep an eye on future development!

`https://aws.amazon.com/neptune/`

Amazon Neptune + RDFox



Amazon Neptune and RDFox: `https://aws.amazon.com/blogs/database/use-semantic-reasoning-to-infer-new-facts-from-your-rdf-graph-by-integrating-rdfox-with-amazon-neptune/`

Graph database/Triplestore benchmarking

- Oracle Database 12c: 1.08 Trillion triples
- AnzoGraph DB: 1.06 Trillion triples
- AllegroGraph: 1.0 Trillion triples
- Virtuoso: 94.2 Billion Triples
- Stardog: 50 Billion triples
- **RDFox**: 19.47 Billion triples
- **GraphDB**: 17 Billion triples
- Apache Jena TBD: 16.7 billion triples.

Numbers may not be up-to-date: `https://www.w3.org/wiki/LargeTripleStores`
`https://medium.com/wallscope/comparison-of-linked-data-triplestores-a-new-contender-c62ae04901d3`
`https://medium.com/wallscope/comparing-linked-data-triplestores-ebfac8c3ad4f`

Laboratory: Hands-on SPARQL

SPARQL: local and remote KG access