SPARQL

- Find matching triples
- Es.
  
  ```
  SELECT * WHERE {
  ?s rdf:type dbo:Film.
  } LIMIT 10
  ```
- ?s is a variable, the result of the query will list all values of ?s that match with a triple
**dbPedia - query**

- go to [http://dbpedia.org/sparql](http://dbpedia.org/sparql)

**note:** some RDF stores have predefined common prefixes, others need to explicitly state all the used prefixes

---

**dbPedia - results**
If are provided more triple patterns they are evaluated in AND

Es:

```sparql
SELECT ?f WHERE {
  ?f a dbo:Film.
} LIMIT 10
```

Retrieves 10 films starred by John Wayne

Es:

```sparql
SELECT ?f WHERE {
  ?f a dbo:Film.
} LIMIT 10
```

Retrieves 10 films starred by John Wayne

Shortcut for `rdf:type`

```sparql
SELECT ?f WHERE {
  ?f a dbo:Film.
} LIMIT 10
```

Retrieves 10 Italian films
Warning!!!

- if in the query you write a **wrong class** name or **property** or **prefix** url NO error is raised, generally it does not provide results.
- Example with 3 errors:
  ```sparql
  PREFIX dbo:<http://dbpedia.org/ontology>
  SELECT * WHERE {
    ?f a dbo:Movie.
  } LIMIT 10
  ```

SPARQL FILTER

- Using FILTER(...condition...) we can filter the returned rows
  ```sparql
  SELECT * WHERE {
    ?f a dbo:Film.
    ?a dbo:birthDate ?bd.
    FILTER(?bd>=xsd:date("1980-01-01"))
  } LIMIT 100
  ```
First 100 film in italian or french

```sparql
SELECT *
WHERE {
  ?f a dbo:Film.
  FILTER(?l="Italian"^^rdf:langString || ?l="French"^^rdf:langString)
} LIMIT 100
```

First 100 film with italian and english title

```sparql
SELECT *
WHERE {
  ?f a dbo:Film.
  FILTER(LANG(?title_it)="it" && LANG(?title_en)="en")
} LIMIT 100
```
with OPTIONAL one or more triple patterns are optional and will not be matched if are not available

```sparql
SELECT *
WHERE {
  ?f a dbo:Film.
  ?f dbo:writer ?w.
  ?w dbo:deathDate ?y.
  OPTIONAL {?f dbo:budget ?b}
}
```

thus we can have rows where column b does not have a value. Without optional the rows without values for b are removed.

---

Example

```sparql
SELECT *
WHERE {
  ?f a dbo:Film.
  OPTIONAL {?f rdfs:label ?it. FILTER(LANG(?it)="it")}
  OPTIONAL {?f rdfs:label ?fr. FILTER(LANG(?fr)="fr")}
}
```

A film may have or not the title in italian or french
How to find film written by alive writers (not dead)?

SELECT * WHERE {
  ?f a dbo:Film.
  ?f dbo:writer ?w.
  FILTER NOT EXISTS { ?w dbo:deathYear ?y }
}

Using ORDER BY rows can be ordered

SELECT * WHERE {
  ?f a dbo:Film.
  ?f dbo:budget ?b
} ORDER BY ?b LIMIT 10

descending...

SELECT * WHERE {
  ?f a dbo:Film.
  ?f dbo:budget ?b
} ORDER BY DESC(?b) LIMIT 10
SPARQL – UNION

- UNION is used to merge the results of two triple patterns:

  ```sparql
  SELECT * WHERE {
    {?f a dbo:Film.} UNION {?f a schema:MusicAlbum}
    ?f foaf:name ?n.
  } ORDER BY ?n
  ```

SPARQL - DISTINCT

- Which classes are on dbpedia?
  ```sparql
  SELECT DISTINCT ?c WHERE {
    ?s rdf:type ?c.
  }
  ```

- Which properties have entities of class Film?
  ```sparql
  SELECT DISTINCT ?p WHERE {
    ?s a dbo:Film.
  }
  ```
We can use the GROUP BY operator in a way similar to SQL

Which classes have more instances?

```sparql
SELECT ?c (COUNT(*) AS ?n) WHERE {
  ?s rdf:type ?c.
} GROUP BY ?c
ORDER BY DESC(?n)
LIMIT 100
```

Can use the same aggregate operators as SQL:
- MAX, MIN, AVG, SUM, COUNT, GROUP_CONCAT, SAMPLE
In the query blank nodes are like variables, can match with any entity URI

```
SELECT * WHERE {
  ?s a dbo:Person.
  ?s dbo:birthPlace [ dbo:country dbr:Italy ].
}
```

or equivalently

```
SELECT * WHERE {
  ?s a dbo:Person.
  ?s dbo:birthPlace _:bn1.
  _:bn1 dbo:country dbr:Italy .
}
```

A blank node used in the data cannot be searched explicitly, it can only be searched via its properties.

However some RDF stores allow to find them, mainly for deletion.
SPARQL - GRAPH

- The "GRAPH pattern" is used to bound triples to a graph
  - GRAPH <http://mygraph.org> {?s a dbo:Place }
  - GRAPH ?g {?s a dbo:Film }
- Which graphs are present and how many triples are containing?
  SELECT ?g (COUNT(*) AS ?n) WHERE {
    GRAPH ?g {?s ?p ?o}
  } GROUP BY ?g
  ORDER BY DESC(?n)

SPARQL – FROM

- It is possible to query data only from specific graphs
  SELECT * FROM <G1> FROM <G2> {
    ...
  }
- \(G_1 \cup G_2\) build the default graph where triples are matched
SPARQL – FROM NAMED

- it is possible to keep the graph
  SELECT * FROM NAMED <G1> FROM NAMED <G2> WHERE {
  ....
}
  The query matches only on the quadruples, so a GRAPH keyword need to be used

  example:
  SELECT DISTINCT ?g
  FROM NAMED <G1> FROM NAMED <G2> WHERE {
    GRAPH ?g {?s ?p ?o}
  }
  returns <G1> and <G2>

SPARQL – SUB QUERY

- Inside a query can be present other queries (bottom-up execution)
- Syntax { SELECT ... WHERE {...} }
- Example:
  SELECT * WHERE {
    {
      SELECT ?a (COUNT(*) AS ?n) WHERE {
        ?a dbo:Film.
      } GROUP BY ?a ORDER BY DESC(?n) LIMIT 5
    }
    ?a dbo:birthDate ?bd.
    FILTER(lang(?name)="en")
  } ORDER BY DESC(?n)
Sub queries can be made in another RDF store (Federated Query)
Syntax: SERVICE <sparql service url> { query }


```sparql
PREFIX dbo:<http://dbpedia.org/ontology/>
SELECT DISTINCT * WHERE {
    ?s a km4c:Municipality.
    ?s foaf:name ?name.
    SERVICE <http://dbpedia.org/sparql> {
        ?sx a dbo:Place.
        ?sx foaf:name ?n.
        ?sx dbo:abstract ?a. FILTER(LANG(?a)="it")
    }
    FILTER(STR(UCASE(?n))=?name)
} ORDER BY DESC(?pp)
LIMIT 1000
```
**SPARQL – Property Paths**

- Property paths allow to navigate properties between two nodes using /,|,^,*,+,*? operators
- **Sequence:**
  ```sparql
  ?f dbo:starring/dbo:birthDate ?bd
  ```
  Equivalent to:
  ```sparql
  ```
- **Choice among two or more properties/path**
  ```sparql
  <S> p1 | p2 <E>
  ```
  Example:
  ```sparql
  ?x dc:title | rdfs:label ?y
  ```
- **Inverse property**
  ```sparql
  <S> ^p1 <E>
  ```
  Equivalent to: `<E> p1 <S>`
  Example:
  ```sparql
  ```
SPARQL – Property Paths

- Property different from a given property
  - `<S> !p1 <E>
  - Example:
    `?a !dbo:birthDate ?nobd`

- Optional property
  - `<S> p1? <E>
  - Example:

SPARQL – Property Paths

- Properties sequence of length >=0
  - `<S> p1* <E>
  - Example:
    `?x foaf:knows* ?y`
  - "Equivalent" to:
    `?x foaf:knows/foaf:knows/.../foaf:knows ?y`

- Properties sequence of length >=1
  - `<S> p1+ <E>
  - Example:
    `?x foaf:knows+ ?y`
  - Equivalent to:
    `?x foaf:knows/foaf:knows* ?y`
Brackets () can be used to group different operators
- `db: Florence (a | !a) + x`

Warning Property Paths cannot contain variables

Can be used to simulate some types of inference:
- `?x rdf:type/rdfs: subClassOf* ?c`
- `?x transitiveProp+ ?y (es. ?x dc: isPartOf+ ?y)`
- `?x reflexAndTransProp* ?y`
- `?x reflexProp? ?y`

Lists can be represented as RDF using properties `rdf: first`, `rdf: rest` and the list `rdf: nil`
- `<#ex> ex: list [ rdf: first "one";
  rdf: rest [ rdf: first "two";
  rdf: rest [ rdf: first "three";
  rdf: rest rdf: nil ] ] ]`
### Query lists

- Property path are useful with recursive structures
- find element with a list starting with "one"
  - `?s ex:list/rdf:first "one"`
- find element with a list containing "two"
  - `?s ex:list/rdf:rest*/rdf:first "two"`
- find element with a list with last element "three"
  - `?s ex:list/rdf:rest* [rdf:first "three"; rdf:rest rdf:nil]`

### Complex Examples

1. Find couples of Film starred from the same three actors.
2. Find film starred from married couples (use `dbo:spouse` property)
3. Find couples of Person born the same day.
solution 1

```sql
select *
    ?f1 a dbo:Film.
    ?f2 a dbo:Film.
    filter(?f1<?f2 && ?a1<?a2 && ?a2<?a3)
} limit 100
```

solution 2

```sql
select *
    ?f a dbo:Film;
    dbo:starring ?a1, ?a2.
}
```
solution 3

```
select * {
    ?p1 a dbo:Person;
    dbo:birthDate ?bd.
    ?p2 a dbo:Person;
    dbo:birthDate ?bd.
}
```

BIND values and functions

- many standard functions can be used to manipulate values:
  - CONCAT(), STRSTARTS(), STRENDSD(), STRBEFORE(), STRAFTER(), STRLEN(), ...
  - isBlank(), isNumeric(), isLiteral(), isUri(), ...
  - ...

- The value of an expression can be associated with a variable:
  - BIND(...expr.. AS ?v)
  - or in the SELECT projection variables
**BIND example**

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name WHERE {
    ?p foaf:givenName ?g ;
    foaf:surname ?s
    BIND(CONCAT(?g, " ", ?s) AS ?name)
}
```

The same can be done

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT (CONCAT(?g, " ", ?s) AS ?name) WHERE {
    ?p foaf:givenName ?g ;
    foaf:surname ?s
}
```

**Geographic search**

- The SPARQL 1.1 recommendation does not specify functions for geographic search
- Each RDF store producer provides its own dialect
- OGC (Open Geospatial Consortium) in 2012 specified classes and properties for the description of geometries in RDF and SPARQL functions for query, however not all producers adopted the specification
**Geographic properties**

- Associate position to a Resource:
  - geo:lat (latitude in decimal degrees)
  - geo:long (longitude in decimal degrees)
  - use the WGS84 reference system
  - Example:
    - `<...> geo:lat "43.123" ; geo:long "10.234".`
- complex geometries are represented using WKT (well known text)
  - "POINT(x y)"^^ogc:wktLiteral
  - "LINESTRING(x1 y1, x2 y2, ...)"^^...
  - "POLYGON((x1 y1,...) (xx1 yy1,...))"^^...

**Geographic indexing**

- Typically wkt literals are indexed with an R-Tree (similar to a B-Tree but with rectangles)
- the index can be used to search in a region
- Virtuoso provide functions
  - bif:st_intersects(?g1, ?g2, ?tolerance)
  - bif:st_distance(?g1, ?g2)
  - bif:st_point(x, y)
  - ...

Virtuoso geographic search example

SELECT * WHERE {
  ?x geo:geometry ?g.
  FILTER(bif:st_intersects(?g,bif:st_point(11.2538,43.7712),1))
  BIND(bif:st_distance(?g,bif:st_point(11.2538,43.7712)) AS ?dist)
} ORDER BY ?dist

SELECT * WHERE {
  ?x geo:geometry ?g.
  BIND(bif:st_distance(?g,bif:st_point(11.2538,43.7712)) AS ?dist)
  FILTER(?dist<1)
} ORDER BY ?dist

GEO SPARQL example

PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/geo sparql/function/>

SELECT ?what WHERE {
  FILTER(geof:within(?geometry, "POLYGON(( -77.089005 38.913574, -77.029953 38.913574, -77.029953 38.886321, -77.089005 38.886321, -77.089005 38.913574 ))"^^geo:wktLiteral))
}
CONSTRUCT

- The SELECT query returns a table of results
- It is possible to generate a graph (a set of triples) rather than a table

CONSTRUCT {
  ... triples with variables ...
} WHERE {
  ... constraints to match variables values ...
}

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT {
  ?x vcard:FN ?name
}
WHERE {
  ?x foaf:name ?name
}
**CONSTRUCT & blank nodes**

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT {
  ?x vcard:N _:v .
  _:v vcard:givenName ?gname .
  _:v vcard:familyName ?fname .
} WHERE {
  { ?x foaf:firstname ?gname } UNION
  { ?x foaf:givenname ?gname } .
  { ?x foaf:surname ?fname } UNION
  { ?x foaf:family_name ?fname } .
}
```

A different blank node for each result

---

**SPARQL query protocol**

- An RDF store exposes a SPARQL endpoint, that can be used to make queries
- Use the HTTP protocol (GET or POST)
- Example:
  - `http://dbpedia.org/sparql?query=...`
  - reply in different formats (Accept: header)
    - json, xml (for a select query)
    - turtle, ntriples, rdf-xml (for a construct query)
**JSON result example**

**query:**

```sql
SELECT ?book ?title WHERE {
}
```

**result:**

```json
{
  "head": {
    "vars": ["book", "title"]
  },
  "results": [
    {
      "bindings": [
        {
          "book": {
            "type": "uri",
            "value": "http://example.org/book/book6"
          },
          "title": {
            "type": "literal",
            "value": "Harry Potter and the Half-Blood Prince"
          }
        },
        {
          "book": {
            "type": "uri",
            "value": "http://example.org/book/book7"
          },
          "title": {
            "type": "literal",
            "value": "Harry Potter and the Deathly Hallows"
          }
        },
        ...
      ]
    }
  ]
}
```

**SPARUL / SPARQL Update**

- Language to insert and delete triples
- **INSERT DATA**

```sparql
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA {
  dc:creator "A.N.Other" .
}
```

- Used when adding few triples
- When adding (millions) triples from files (ntriples, turtle, ...)
  each store has its own way
DELETE & INSERT DATA

PREFIX dc: <http://purl.org/dc/elements/1.1/>

DELETE DATA {
    GRAPH <http://example/bookStore> {
        <http://example/book1> dc:title "Fundamentals of Compiler Design"
    }
};

PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA {
    GRAPH <http://example/bookStore> {
        <http://example/book1> dc:title "Fundamentals of Compiler Design"
    }
}

this is the only way to update a triple, delete and then insert

DELETE & INSERT

- allows to delete and insert triples on the basis of a query patterns in a WHERE clause

PREFIX foaf: <http://xmlns.com/foaf/0.1/>

WITH <http://example/addresses>
DELETE {
    ?person foaf:givenName 'Bill'
}
INSERT {
    ?person foaf:givenName 'William'
}
WHERE {
    ?person foaf:givenName 'Bill'
}

Changes the names of 'Bill' to 'William', in general the DELETE and INSERT sections are optional
**INSERT**

PREFIX foaf: <http://xmlns.com/foaf/0.1/>

**INSERT**

```{ GRAPH <http://example.org/friends> {
  ?p1 foaf:knows ?p3
}
WHERE {
} ```

**Insert & inference**

- The INSERT may be used as inference rules to be run after each data update
- Example:
  ```
  INSERT {
  } WHERE {
      ?c1 rdfs:subClassOf ?c2.
  }
  ```
Insert & federated query

- can be used to get and manipulate data from other

Other graph commands

- DROP GRAPH <...>
- COPY GRAPH <G1> TO GRAPH <G2>
- MOVE GRAPH <G1> TO GRAPH <G2>
- ADD GRAPH <G1> TO GRAPH <G2>
SPARQL REST interface

- standardized REST interface to interact with a RDF store and manage the GRAPHs.
- GET ...url...?graph=...graph uri...
  - retrieves the content of the graph (Accept header to state the format)
- PUT ...url...?graph=...graph uri...
  - insert or replace the data in the graph
- POST ...url...?graph=...graph uri...
  - add the data provided to the graph
- DELETE ...url...?graph=...graph uri...
  - delete the graph content