SPARQL
An RDF Query Language
SPARQL

- SPARQL is a recursive acronym for **S**PARQL **P**rotocol **A**nd **R**df **Q**uery **L**anguage
- SPARQL is the SQL for RDF
- Example query suitable for DBpedia

```
# find countries and their languages
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT * WHERE {
  ?country a dbo:Country;
}
LIMIT 10
```
Several RDF query languages were developed prior to SPARQL

W3C RDF Data Access Working Group (DAWG) worked out SPARQL 2005-2008

Became a W3C recommendation in Jan 2008

SPARQL 1.1 (2013) is the current standard

Implementations for multiple programming languages available
Typical Architecture

SPARQL endpoint receives queries and requests via HTTP from programs or GUIs, accesses associated RDF triple store and returns result, e.g., data.
Some SPARQL endpoints

There are many public endpoints, e.g.

- Dbpedia: https://dbpedia.ort/sparql/
- Wikidata: https://query.wikidata.org/sparql
- DBLP: https://dblp.l3s.de/d2r/sparql
- See W3C’s list of currently alive SPARQL endpoints

It’s not hard to set up your own, e.g.

Some endpoints offer their own SPARQL GUI you can use to enter ad hoc queries. They may use the same URL as the REST interface and rely on the protocol to know when it’s a person and when a query:

- Dbpedia: [http://dbpedia.org/sparql/](http://dbpedia.org/sparql/)
- Wikidata: [https://query.wikidata.org/](https://query.wikidata.org/)
- DBLP: [https://dblp.l3s.de/d2r/snorql/](https://dblp.l3s.de/d2r/snorql/)
You can also access or run a general SPARQL GUI that can talk to any SPARQL endpoint. A nice example is YASGUI, which has a public resource: [https://yqagui.org/](https://yqagui.org/) and is available to download. Another open-source GUI is Twinkle.
YASGUI: Yet Another SPARQL GUI

```sparql
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT * WHERE {
}
LIMIT 10
```

<table>
<thead>
<tr>
<th>country</th>
<th>lang</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://dbpedia.org/resource/Arab_League">http://dbpedia.org/resource/Arab_League</a></td>
<td><a href="http://dbpedia.org/resource/Arabic_language">http://dbpedia.org/resource/Arabic_language</a></td>
</tr>
</tbody>
</table>
SPARQL query structure

- **Prefix declarations**, for abbreviating URIs
- **Dataset definition**, stating what RDF graph(s) are being queried
- A **result clause**, says what information to return from the query
- The **query pattern**, says what to query for in the underlying dataset
- **Query modifiers**, slicing, ordering, and otherwise rearranging query results

```sparql
# prefix declarations
PREFIX foo: <http://example.com/resources/>
...
# optional named graph source
FROM ...
# result clause (select, ask, update...)
SELECT ...
# query pattern
WHERE { ... }
# query modifiers
ORDER BY ...
LIMIT 100
```
Basic SPARQL Query Forms

- **SELECT**
  Returns all, or a subset of, the variables bound in a query pattern match

- **ASK**
  Returns a boolean indicating whether a query pattern matches or not

- **DESCRIBE**
  Returns an RDF graph describing resources found

- **CONSTRUCT**
  Returns an RDF graph constructed by substituting variable bindings in a set of triple templates
A Query: Maryland Cities

# find URIs for cities in Maryland
PREFIX yago: <http://dbpedia.org/class/yago/>
SELECT * WHERE {
  ?city a yago:WikicatCitiesInMaryland
}

To use this query, we need to know:

- What endpoint (URL) to send it to
- How we want the results encoded (JSON, XML, ...)
- ... other parameters ...

These are set in GUI or your program:

- Except for the endpoint, all have defaults

Can even query with the unix curl command:

Maryland Cities and population

PREFIX yago: t<http://dbpedia.org/class/yago/>t
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT * WHERE {
  ?city a yago:WikicatCitiesInMaryland;
  dbo:populationTotal ?population .
}
SELECT ?city ?name ?population WHERE {
    ?city a yago:WikicatCitiesInMaryland;
    dbo:populationTotal ?population ;
    rdfs:label ?name .
}
select (str(?name) as ?name) ?population where {
  ?city a yago:WikicatCitiesInMaryland;
    dbo:populationTotal ?population;
    rdfs:label ?name .

  FILTER (LANG(?name) = "en")
}
# sort results by population

PREFIX yago: http://dbpedia.org/class/yago/
PREFIX dbo: <http://dbpedia.org/ontology/>

select str(?name) ?population where {
  ?city a yago:WikicatCitiesInMaryland;
  dbo:populationTotal ?population;
  rdfs:label ?name .
  FILTER (LANG(?name) = "en")
}

ORDER BY DESC(?population)
Wait, where’s Catonsville?

- Maryland’s government focused on counties
- Catonsville is not considered a city – it has no government
- We need another category of place
  - Census designated place? Populated Place?
- Populated places include counties & regions, so let’s go with census designated place
PREFIX yago: <http://dbpedia.org/class/yago/>
PREFIX dbo: http://dbpedia.org/ontology/
PREFIX dbr: <http://dbpedia.org/resource/>

select str(?name) ?population where {
    {?city dbo:type dbr:Census-designated_place;
     dbo:isPartOf dbr:Maryland .}

    UNION

    {?city a yago:WikicatCitiesInMaryland .}
}

?city dbo:populationTotal ?population;
    rdfs:label ?name .

FILTER (LANG(?name) = "en")
}

ORDER BY DESC(?population)
Now we have some duplicate entries

This happens because:
- Some “cities” are just in WikicatCitiesInMaryland
- Some are just in Census-designated_places
- Some are in both

SPARQL’s procedure finds all ways to satisfy a query, and for each one, records the variable bindings

We add DISTINCT to get SPARQL to remove duplicate bindings from the results
PREFIX yago: <http://dbpedia.org/class/yago/>
PREFIX dbo: http://dbpedia.org/ontology/
PREFIX dbr: <http://dbpedia.org/resource/>

select DISTINCT str(?name) ?population where {
    {?city dbo:type dbr:Census-designated_place;
        dbo:isPartOf dbr:Maryland .}

    UNION

    {?city a yago:WikicatCitiesInMaryland .}

    ?city dbo:populationTotal ?population;
        rdfs:label ?name .

    FILTER (LANG(?name) = "en")
}

ORDER BY DESC(?population)
Some cities are missing 😞

- Experimentation with query showed there are 427 entities in MD that are either census designated places or cities.
- Only get 411 because nine have no population and one has neither a population nor a label.
  - Typical of a large and somewhat noisy knowledge graph created from crowdsourced data.
- SPARQL’s OPTIONAL directive to the rescue.
select DISTINCT str(?name) ?population where {
  {?city dbo:type dbr:Census-designated_place;
   dbo:isPartOf dbr:Maryland .}

UNION

  {?city a yago:WikicatCitiesInMaryland .}

OPTIONAL {?city dbo:populationTotal ?population.}
  OPTIONAL {?city rdfs:label ?name . FILTER (LANG(?name) = "en")}
}

ORDER BY DESC(?population)
Handling queries with many results

- Endpoints typically have limits on a query’s runtime or the number of results it can return.
- You can use the LIMIT and OFFSET query modifiers to manage large queries.
- Suppose we want to find all of the types that DBpedia uses:
  
  ```sparql
  SELECT distinct ?type WHERE {
    ?x a ?type . }
  ```

- DBpedia’s public endpoint limits queries to 10K results.
Get the first 10K

[SPARQL Query]

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT distinct ?type WHERE {
  ?x a ?type .
}
```

[Table]

<table>
<thead>
<tr>
<th>type</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="http://www.openlinksw.com/schemas/virtrdf#QuadMapFormat">http://www.openlinksw.com/schemas/virtrdf#QuadMapFormat</a></td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.openlinksw.com/schemas/virtrdf#QuadStorage">http://www.openlinksw.com/schemas/virtrdf#QuadStorage</a></td>
</tr>
<tr>
<td>3</td>
<td><a href="http://www.openlinksw.com/schemas/virtrdf#array-of-QuadMapFormat">http://www.openlinksw.com/schemas/virtrdf#array-of-QuadMapFormat</a></td>
</tr>
</tbody>
</table>
Get the second 10K
from SPARQLWrapper import SPARQLWrapper, JSON
default_endpoint = "http://dbpedia.org/sparql"

query = """"SELECT DISTINCT ?class WHERE {{?x a ?class}} LIMIT {LIM} OFFSET {OFF}"""

def getall(query, endpoint=default_endpoint):
    limit = 10000
    offset = total = 0
    found = limit
    tuples = []
    sparql = SPARQLWrapper(endpoint)
    sparql.setReturnFormat('json')
    while found == limit:  # keep going until we don't get limit results
        q = query.format(LIM=limit, OFF=offset)
        sparql.setQuery(q)
        results = sparql.query().convert()
        found = 0
        for result in results["results"]["bindings"]:  
            found += 1
            tuples.append(tuple([str(v['value']) for v in result.values()])))
        print('Found', found, 'results')
        total = total + found
        offset = offset + limit
    return tuples
An ASK query returns True if it can be satisfied and False if not

Was Barack Obama born in the US?

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>

ask WHERE {
    {dbr:Barack_Obama dbo:birthPlace dbr:United_States}
    UNION
    {dbr:Barack_Obama dbo:birthPlace ?x .
     ?x dbo:isPartOf*/dbo:country dbr:United_States }
}
“Describe ?x” means “tell me everything you know about ?x"

Example: Describe Alan Turing ...

```
DESCRIBE <http://dbpedia.org/resource/> Alan_Turing>
-- or --
PREFIX dbr: <http://dbpedia.org/resource/>
DESCRIBE dbr:Alan_Turing
```

- Returns a collection of ~1500 triples in which dbr:Alan_Turing is either the subject or object
The DAWG did not reach a consensus on what describe should return.

Possibilities include:

- All triples where the variable bindings are mentioned.
- All triples where the bindings are the subject.
- Something else.

What is useful might depend on the application or the amount of data involved.

So it was left to the implementation.
Describe the film “Double Indemnity”

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

describe ?x WHERE {
  ?x a dbo:Film; foaf:name ?filmName .
  FILTER (STR(?filmName) = "Double Indemnity")
}

Returns a collection of ~500 triples
Describe can return triples about multiple entities

Describe films directed by Billy Wilder

PREFIX dbo: http://dbpedia.org/ontology/
PREFIX dbr: <http://dbpedia.org/resource/>
describe ?x WHERE {
  ?x a dbo:Film; dbo:director dbr:Billy_Wilder.
}

Returns a collection of ~8400 triples about the 27 films he directed
Describe can return triples about multiple entities, but you can limit the number

Describe films directed by Billy Wilder

PREFIX dbo: http://dbpedia.org/ontology/
PREFIX dbr: <http://dbpedia.org/resource/>
describe ?x WHERE {
  ?x a dbo:Film; dbo:director dbr:Billy_Wilder.
} LIMIT 1

Returns a collection of ~500 triples about just one film, The Apartment.
Construct queries return graphs as results, e.g., film directors and the actors they’ve directed.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX ex: <http://example.org/>
CONSTRUCT {?director ex:directed ?actor}
WHERE {?film a dbo:Film;
  dbo:director ?director;
  dbo:starring ?actor}
```

Returns a graph with ~21,000 triples
On construct

- Having a result form that produces an RDF graph is a good idea.
- It enables one to construct systems by using the output of one SPARQL query as the data over which another query works.
- This kind of capability was a powerful one for relational databases.
Construct queries return graphs as results, e.g., film directors and the actors they’ve directed.

```sparql
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX ex: <http://example.org/>
CONSTRUCT {?director ex:directed ?actor}
WHERE {?film a dbo:Film;
    dbo:director ?director;
    dbo:starring ?actor}
```

- Returns a graph with ~21,000 triples
Construct query (3)

- Actors and directors or producers they've worked for
  
  ```sparql
  PREFIX dbo: <http://dbpedia.org/ontology/>
  PREFIX ex: <http://example.org/>
  Construct {?actor ex:workedFor ?directorOrProducer}
  WHERE {
    ?film a dbo:Film;
    dbo:director|dbo:producer ?directorOrProducer;
    dbo:starring ?actor}
  ```

- Returns a graph with ~31,000 triples

SPARQL 1.1 allows using alternative properties separated by vertical bar
Example: finding missing inverses

- DBpedia is missing many inverse relations, including more than 10k missing spouse relations.
- This creates a graph of all the missing ones, which can be added back to the KG via UPDATE ADD

```
PREFIX dbo: <http://dbpedia.org/ontology/>
CONSTRUCT { ?p2 dbo:spouse ?p1. }
FILTER NOT EXISTS {?p2 dbo:spouse ?p1}}
```

- Not the **NOT EXISTS** operator that succeeds iff its graph pattern is not satisfiable.
RDF Named graphs

- Having multiple RDF graphs in a single document/repository and naming them with URIs
- Provides useful additional functionality built on top of the RDF Recommendations
- SPARQL queries can involve several graphs, a background one and multiple named ones, e.g.:

  ```sparql
  SELECT ?who ?g ?mbox
  FROM <http://example.org/dft.ttl>
  FROM NAMED <http://example.org/alice>
  FROM NAMED <http://example.org/bob>
  WHERE
  { ?g dc:publisher ?who .
    GRAPH ?g { ?x foaf:mbox ?mbox } }
  ```
**Simple insert**

```
```

**Simple delete**

```
```

**Combine the two for a modification, optionally guided by the results of a graph pattern**

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

DELETE { ?person foaf:givenName 'Bill' }
INSERT { ?person foaf:givenName 'William' }
WHERE { ?person foaf:givenName 'Bill' }
```
SPARQL 1.1 added many aggregation operators, like count, min, max, ...

Generally used in the results specification

```
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT (COUNT(?film) AS ?numberOfFilms)
WHERE {?film a dbo:Film .}
```

This finds 129,980 films
GROUP BY breaks the query's result set into groups before applying the aggregate functions.

Find BO’s properties and group them by property and find the number in each group.

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?p (COUNT(?p) as ?number)
WHERE { dbr:Barack_Obama ?p ?o }
GROUP BY ?p ORDER BY DESC(count(?p))
```
This query adds inverse spouse relations that don’t already exist:

```sparql
PREFIX dbo: <http://dbpEDIA.org/ontology/>
INSERT { ?p2 dbo:spouse ?p1. }
    FILTER NOT EXISTS {?p2 dbo:spouse ?p1}}
```

- **SPIN** and **SHACL** are systems to represent simple constraint & inference rules that are done by sparql
- A big feature is that the rules are represented in the graph
SPARQL 1.1 Additions

- SPARQ 1.1 added many more features ...
  - Subqueries
  - Negation: MINUS
  - Federated queries that access multiple endpoints

- Any data you want to extract from an rdf graph, can probably be returned by one query

- Search the web for SPARQL tricks or this book