Semantic Web e Linguaggi di interrogazione di dati RDF

Ing. Irina Trubitsyna

Outline

- Semantic Web
- RDF
- SPARQL

World Wide Web

- The World Wide Web was invented by Sir Tim Berners-Lee in 1989. The key technology of the original web—from an end user's point of view, anyway—was the hyperlink. A user could click on a link and go to the document identified in that link.
- The great advantage of Web 1.0 was that it abstracted away the physical storage and networking layers involved in information exchange between two machines.
- This breakthrough enabled documents to appear to be directly connected to one another. Click a link and you're there—even if that link goes to a different document on a different machine on another network on another continent!

Semantic Web

- The Semantic Web represents the next major evolution in connecting information. It enables data to be linked from a source to any other source and to be understood by computers so that they can perform increasingly sophisticated tasks on our behalf.
- In the same way that Web 1.0 abstracted away the network and physical layers, the Semantic Web abstracts away the document and application layers involved in the exchange of information.
- The Semantic Web connects facts, so that rather than linking to a specific document or application, you can instead refer to a specific piece of information contained in that document or application. If that information is ever updated, you can automatically take advantage of the update.

Differences

- The fundamental difference between Semantic Web technologies and other technologies related to data (such as relational databases or the World Wide Web itself) is that the Semantic Web is concerned with the meaning and not the structure of data.
- This fundamental difference engenders a completely different outlook on how storing, querying, and displaying information might be approached.
- Some applications, such as those that refer to a large amount of data from many different sources, benefit enormously from this feature. Others, such as the storage of high volumes of highly structured transactional data, do not.

Primarily Technical Standards

- **RDF (Resource Description Framework):** The data modeling language for the Semantic Web. All Semantic Web information is stored and represented in the RDF.
- **SPARQL (SPARQL Protocol and RDF Query Language):** The query language of the Semantic Web. It is specifically designed to query data across various systems.
- **OWL (Web Ontology Language):** The schema language, or knowledge representation (KR) language, of the Semantic Web. OWL enables you to define concepts so that these concepts can be reused as much and as often as possible. Each concept should be carefully defined so that it can be selected and assembled in various combinations with other concepts as needed for many different applications and purposes.
- Though there are other standards sometimes referenced by Semantic Web literature, these are the foundational three.
- One way to differentiate a Semantic Web application vs. any other application is the usage of those three technologies.

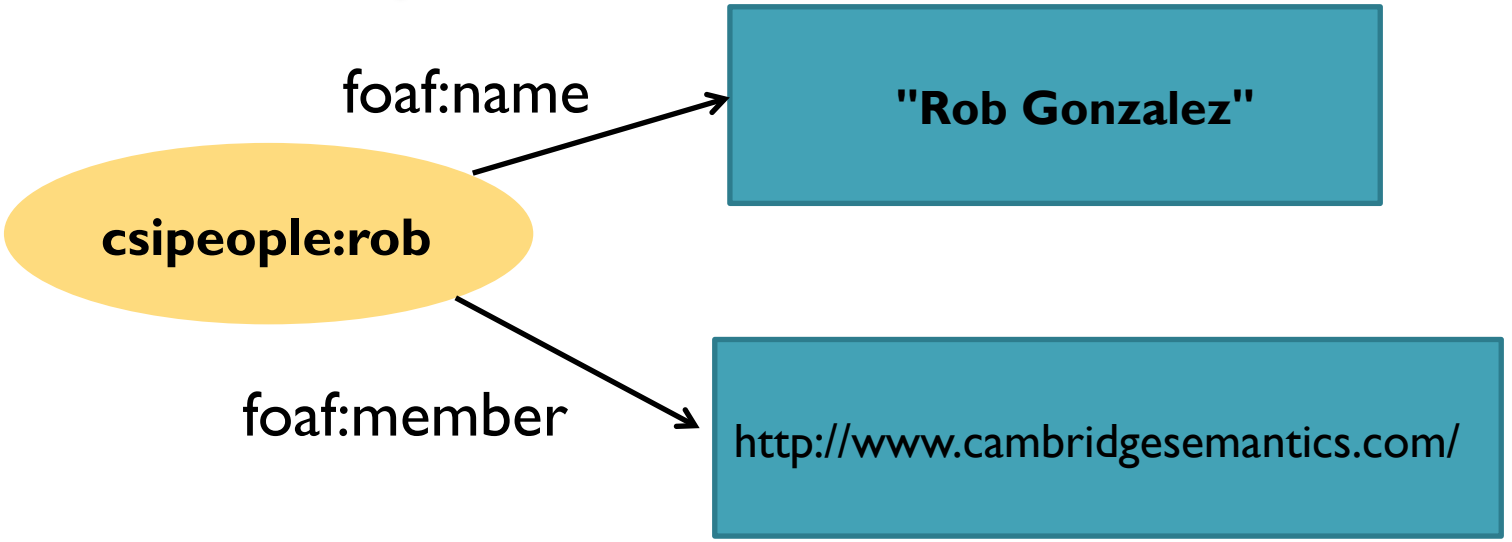
RDF

- RDF is the data model of the Semantic Web: all data in Semantic Web technologies is represented as RDF.
  - If you store Semantic Web data, it's in RDF.
  - If you query Semantic Web data (typically using SPARQL), it's RDF data.
  - If you send Semantic Web data to your friend, it's RDF.

RDF

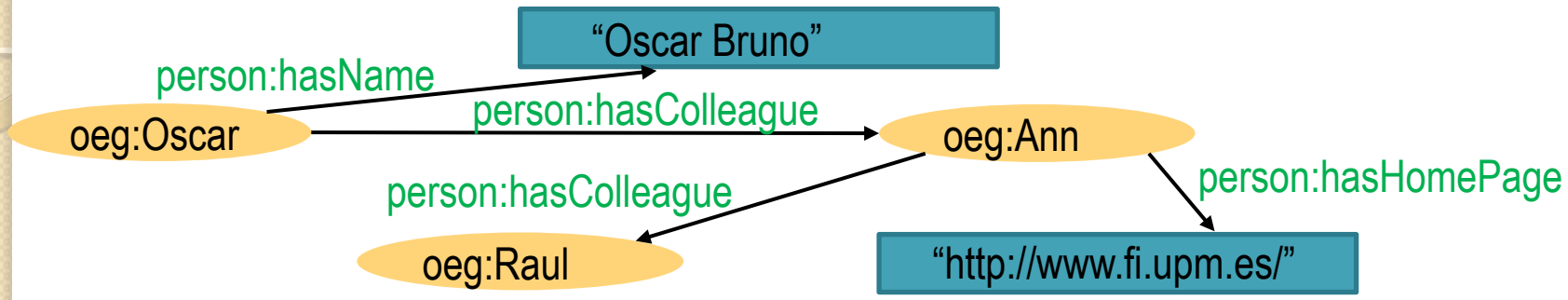
- All data in the Semantic Web is represented in RDF, including schema describing RDF data.
- RDF is not like the tabular data model of relational databases. Nor is it like the trees of the XML world.
- **RDF is a labeled directed graph**.

RDF Graph



- There are three kinds of nodes in an RDF directed graph:
  - **Resource nodes** (represented by ovals) : A resource is anything that can have things said about it. It's easy to think of a resource as a thing vs. a value.
  - **Literal nodes** (represented by rectangles): The term literal is a fancy word for value.
  - **Blank nodes**. A blank node is a resource without a URI.
- Edges can go from any resource to any other resource, or to any literal, with the only restriction being that edges can't go from a literal to anything at all.

RDF Graph



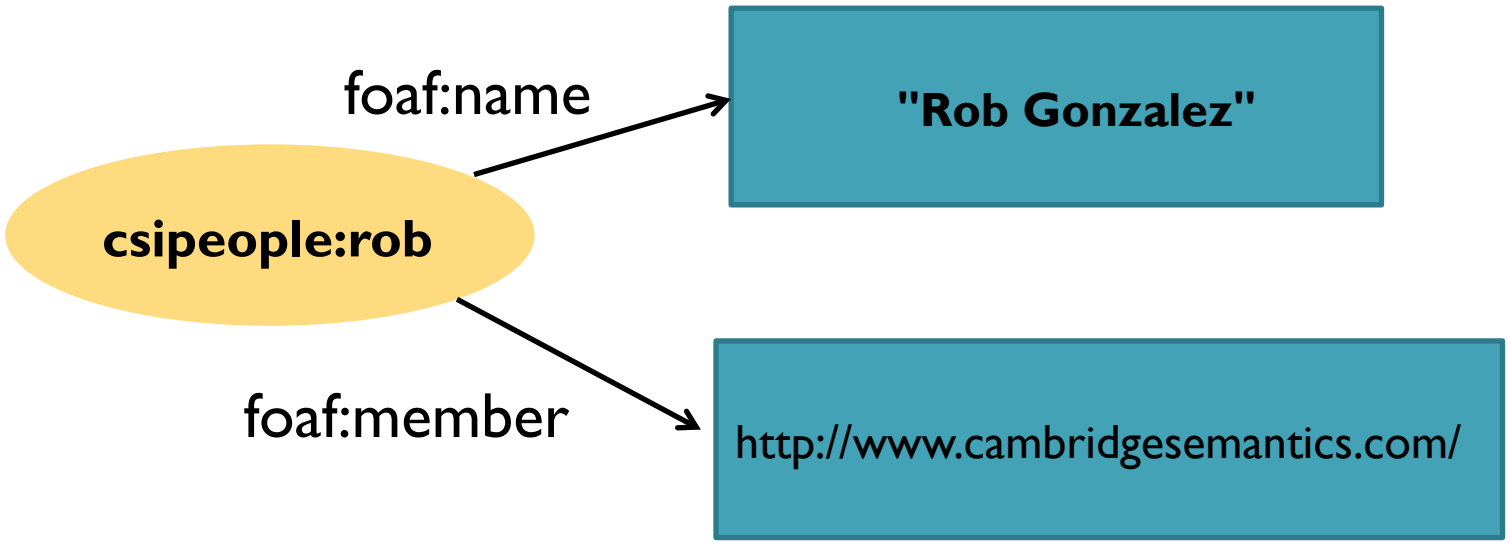
- Anything in RDF can be connected to anything else simply by drawing a line.
- Creating a new thing is as easy as drawing an oval.
- The ability to connect anything together, any time you want, is revolutionary. It's like hyperlinking on the Web, but for any data you have!
- This linking between things is the fundamental capability of the Semantic Web, and is enabled by the URI.

Flexibility of RDF model

- Semantic Web technologies are more flexible than other technologies (XML, relational databases, BI cubes, etc.)
- If you want to connect two things in a relational database you have to add foreign keys to tables (or, if you have a many-to-many relationship, create join tables), etc. If you want to link things between databases, you need an ETL job. It's just not easily done.
- If you consider the XML world, the same thing is true. Connecting things within an XML document is possible, if tedious. Connecting things between XML documents requires real work. Unless you're one of the very few who just loves XSLT, you're not doing that very often.

URI - Universal Resource Identifier

- Instead of making ad hoc IDs for things within a single database (think primary keys), in the Semantic Web we create universal identities for things that are consistent across databases. This enables us to create linkages between all things.
- In RDF, resources and edges are URIs.



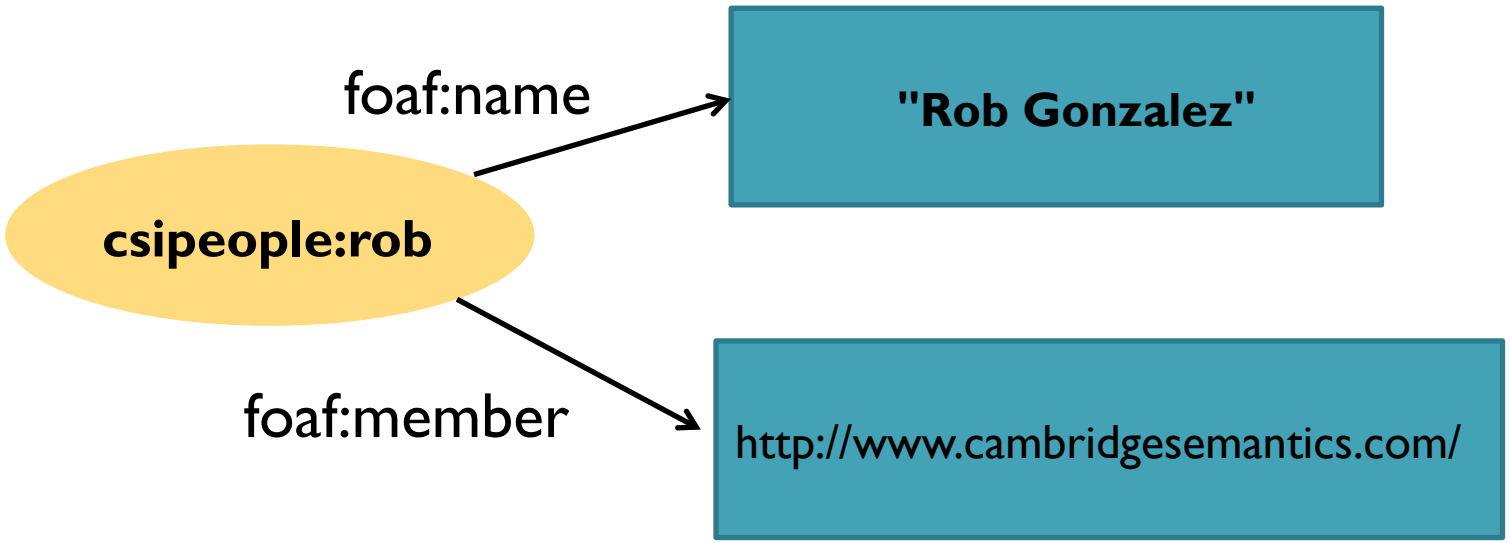
csipeople:rob shorthand for www.cambridgesemantics.com/people/about/rob
foaf:member shorthand for http://xmlns.com/foaf/0.1/member
foaf:name shorthand for http://http://xmlns.com/foaf/0.1/name

RDF Vocabularies

- There is a major problem with the concept of universality presented above. It's impossible to get everyone everywhere to agree on a single label for every specific thing that ever was, is, or shall be.
- What we mean by an RDF Vocabulary is essentially the set of URIs for the edges that make up RDF graphs. The edges are what relates the things in graph, and are what give it meaning. Using specific URIs is like speaking in a specific language—hence the term vocabulary.
- In order for two Semantic Web applications to share data, they must
  - agree on a common vocabulary (created by means of RDFS or OWL) or
  - translate RDF written in one vocabulary to another vocabulary (using SPARQL).
- The Semantic Web technologies were built under the assumption that different people in different applications written for different purposes at different times would create related concepts that overlap in any number of ways, and therefore there are provisions and methods to make it all work together with little effort. There is no such provision in the XML or relational database worlds.
- This same universal identity conundrum also happens for resources. Semantic Web offers very simple ways to merge identical concepts so that they appear as one universally.

Statement or triple

- Statements or triples represent graph edges are 3-tuples of the form **(subject, predicate, object)**.
- RDF graphs therefore are simply collections of triples.
- An RDF database is often called a triple store for this reason.



(csipeople:rob, foaf:name, "Rob Gonzalez")
(csipeople:rob, foaf:member, http://www.cambridgesemantics.com/)

Data segmentation problem

- It is very difficult to deal with large amounts of triples for application development. There are lots of reasons that you would want to segment different subsets of triples from each other:
  - simplified access control,
  - simplified updating,
  - trust, etc
- At first the community tried using *reification* to solve this data segmentation problem, but today everyone has converged on using *named graphs*.

Named Graphs and Quads

- A named graph is simply a collection of RDF statements that has a name which is a URI.
- Modern triple stores all support named graphs, and they are built into SPARQL 1.1, the latest SPARQL query language specification.
- When referring to a triple in a named graph, you would often use 4-tuple notation instead of 3-tuple notation:

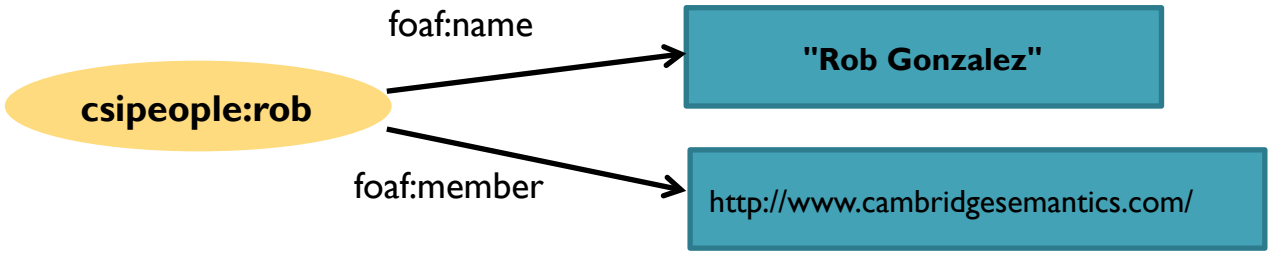
  (named graph, subject, predicate, object)
- For this reason, a triple store that supports named graphs is often called a quad store.
- Looking at the 4-tuples, it's pretty obvious that the same statement can exist in multiple named graphs. This is by design and is a very important feature.
- By organizing the statement into named graphs, a Semantic Web application can implement access control, trust, data lineage, and other functionality very cleanly.

Serializations for RDF data

RDF is an *abstract* data model. There is not a single way to represent it. Instead there are several valid serializations for RDF data, including:

- **RDF/XML.** This is simply RDF represented as valid XML. This was originally proposed and used due to the plethora of existing tools that could parse and store XML. RDF/XML is verbose and somewhat difficult to read and write as a human, though it can be read and written by just about any RDF tool, so you'll see it around. It's usually not the best serialization to use.
- **N-Triples.** N-Triples is a very basic RDF serialization. Its key feature is that only one triple exists per line so that it's very quick to parse and so that Unix command-line tools can easily manipulate it. It's also highly compressible, so large, public RDF sources like *DBpedia* often publish data in N-Triples form.
- **Turtle.** If you're writing RDF today, you're probably writing it in Turtle. Turtle is significantly more compact than RDF/XML, more readable than N-Triples, and lacks the first-order logic extensions from Notation3. Furthermore, the SPARQL query language expresses RDF queries in almost exactly the same way.
- **TriG.** TriG is Turtle but with support for named graphs. It's the de facto standard for serializing RDF with named graphs.
- **RDFa (RDF embedded in HTML).** You can embed RDF data within normal web pages by using RDFa. This is a very powerful technique that has been used by major companies such as Best Buy.
- **Notation3 (N3 for short).** This is a largely legacy serialization that was originally proposed by Tim Berners-Lee in 1998. It extended RDF with a form of first-order logic that was never very popular.

Examples of Serializations



N-Triples:

<http://www.cambridgesemantics.com/people/about/rob> <http://xmlns.com/foaf/0.1/name> "Rob Gonzalez" .
<http://www.cambridgesemantics.com/people/about/rob> <http://xmlns.com/foaf/0.1/member > <http://www.cambridgesemantics.com/> .

Turtle:

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix csi: <http://www.cambridgesemantics.com/> .
@prefix csipeople: <http://www.cambridgesemantics.com/people/about/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
csipeople:rob foaf:name "Rob Gonzalez"^^xsd:string .
csipeople:rob foaf:member csi: .

A set of statements about a single resourcen can be combined:

csipeople:rob
foaf:name "Rob Gonzalez"^^xsd:string ;
foaf:member csi: .

TriG and Named Graphs

- When using named graphs, TriG is the de facto serialization. It's the same as Turtle except that statements in a single graph are grouped with {}.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix csi: <http://www.cambridgesemantics.com/> .
@prefix csipeople: <http://www.cambridgesemantics.com/people/about/>
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix : <http://www.cambridgesemantics.com/semantic-university> .

:myGraph {
      csipeople:rob foaf:name "Rob Gonzalez"^^xsd:string .
      csipeople:rob foaf:member csi: .
}
```

- This trivial example puts all the statements in the document into a single named graph :myGraph.
- Again, like all things in RDF, :myGraph is a URI.
- TriG documents have the extension .trig.

What to use for predicates?

- One thing you can do is to simply make up your own predicates, just as you made up your own resources. There are valid arguments for creating new vocabularies even when there exist ones that you could use.
- Vocabularies are defined using RDFS (RDF Schema) or OWL (Web Ontology Language).
- To get started, a few simple, common vocabularies include:
  - **DC (Dublin Core):** standard, general-purpose vocabulary for describing resource metadata such as titles and creators.
  - **FOAF (Friend of a Friend):** lots of vocabulary to relate people to each other and to organizations.
  - **GoodRelations:** ecommerce vocabulary.

Triple Stores

- Now that you can create RDF, no doubt you want to store it somewhere. That's where a triple store comes in.
- There are a variety of triple stores available that differ in performance, scalability, platform support, APIs and features. For example:
  - Most support transactions, but granularity varies.
  - Most support access control, but granularity varies.
  - Some focus on Big Data and scale over clusters of machines, others do not.
  - Some have analytic capabilities for doing aggregations and mathematical calculations to create charts and graphs. Most do not.
- So there is no "best" store, and no clear "Oracle of RDF" for now. In fact, both Oracle and IBM DB2 have basic support for RDF today in their flagship products.
- So which to choose?
- Wikipedia lists a variety of stores, including Apache Jena, Sesame, Virtuoso, AllegroGraph, BigData, OWLIM, and more. For getting started just look around and pick the one that you're most excited about. Make sure to look out for good getting started guides, but otherwise don't stress too much about your choice. When you get further along and have a better idea what features matter for your purposes, you can always switch to another store. That's the beauty of RDF; it's standard!

Triple stores

- For smaller projects or for embedding RDF technologies into a larger application a standalone triple store might be exactly what you need.
- For enterprise applications I would recommend going with a larger platform that provides more tooling and high-level data management capabilities than pure triple stores.
- Think about named graphs for a minute. Let's say that you have an RDF database with a *billion* triples. Questions arise:
  - How do you manage transactions (what do you lock when adding or removing a single triple? Think about a block of triples representing a profile using FOAF, for example.)?
  - Access control (who can see what triples)?
  - Versioning?
  - Named graphs are a good first step towards this, but they complicate queries. It's expensive to have your SPARQL queries go over an entire database, so you need some kind of heuristic determining which triples go into which named graphs. How do you do this?
- The big enterprise platforms all have some answer to these questions, and, in general, for real applications you don't want to be reinventing the wheel here.

SPARQL

- SPARQL (pronounced "sparkle") is the query language for the Semantic Web.
- Along with RDF and OWL, it is one of the three core technologies of the Semantic Web.
- SPARQL is a recursive acronym, which stands for SPARQL Protocol and RDF Query Language.
- Data exposed via SPARQL *on any server* can be queried by any SPARQL client. This is a fundamental difference between SPARQL and other query languages, such as SQL, which assume that all data being queried is local and conforms to a single model.
- With SPARQL—and especially when using CONSTRUCT—data from multiple places can be combined dynamically, as needed, to create new forms of information.

SPARQL fundamentals

- At its most basic, a SPARQL query is an RDF graph with variables.

  ex:juan foaf:name "Juan Sequeda" .
  ex:juan foaf:based_near ex:Austin .

- Now consider a version of the previous RDF graph that has variables instead of values:

  ?x foaf:name ?y .
  ?x foaf:based_near ?z .

- A basic SPARQL query is simply a graph pattern with some variables. Data that is returned via a query is said to match the pattern. To get the actual RDF graph we showed above we bind

  ?x to ex:juan,

  ?y to "Juan Sequeda" and

  ?z to ex:Austin.

SPARQL query

- The following SPARQL query has all the major components from SPARQL:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM <http://example.com/dataset.rdf>
WHERE {
 ?x foaf:name ?name .
}
ORDER BY ?name
```

- The PREFIX keyword describes prefix declarations for abbreviating URIs.
- The SELECT keyword returns data matching some conditions. In particular, SELECT queries return data represented in a simple table, where each matching result is a row, and each column is the value for a specific variable.
- The FROM keyword defines the RDF dataset which is being queried. There is an optional clause, FROM NAMED, which is used when you want to query a named graph.
- The WHERE clause specifies the query graph pattern to be matched. This is the heart of the query. A graph pattern, as mentioned above, is, in essense, RDF with variables.
- Finally, ORDER BY is one of the several possible solution modifiers, which are used to rearrange (riorganizzare) the query results. Other solution modifiers are LIMIT and OFFSET.

Return Clauses

- **ASK** queries check if there is at least one result for a given query pattern. The result is true or false.
- **DESCRIBE** queries returns an RDF graph that describes a resource. The implementation of this return form is up to each query engine, so you won't see it used nearly as often as the other return clauses.
- **CONSTRUCT** queries returns an RDF graph that is created from a template specified as part of the query itself. That is, a new RDF graph is created by taking the results of a query pattern and filling in the values of variables that occur in the construct template.
  - CONSTRUCT is used to transform RDF data (for example into a different graph structure and with a different vocabulary than the source data).
  - CONSTRUCT queries are useful if you have RDF data that was automatically generated and would like to transform it using well-known vocabularies, or if you have RDF data using vocabulary from one ontology but need to translate it to another ontology.
  - Translation using CONSTRUCT is relatively cheap.

Query 1

This query returns all of the URIs that identify cities that are of type "Cities in Texas".

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas>
}
```

Result: a table containing a rather long list of cities in Texas.

Query 2a

This query returns the cities that are of type "Cities in Texas" as well as their total populations.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>

SELECT *WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> .
    ?city dbp:populationTotal ?popTotal .
}
```

Notice that the variable ?city is used as the subject of both triples in the query, matching two statements on a single resource.

Results page is largely the same as the first query, but there is now a new column with the extra information requested.

Query 2b

This query returns the cities that are of type "Cities in Texas" as well as their total populations.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>

SELECT *WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
          dbp:populationTotal ?popTotal .
}
```

SPARQL uses the Turtle syntax so the query is the same:

Query 3

This query returns the cities that are of type "Cities in Texas" with their total populations and metro populations.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>

SELECT *WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
          dbp:populationTotal ?popTotal ;
          dbp:populationMetro ?popMetro .

}
```

- Now the size of the results has shrunk dramatically from the previous queries! In fact, by asking for the additional information we are putting an implicit restriction on the query.
- Specifically, the query will return results *only* for cities that have values for both ?popTotal and ?popMetro. Cities that only have ?popTotal but *not* ? popMetro do not show up anymore.

Query 4

This query returns the cities that are of type "Cities in Texas" and their total population and optionally the metro population, *if it exists*.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>

SELECT *WHERE {
   ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
         dbp:populationTotal ?popTotal ;
         dbp:populationMetro ?popMetro .
         OPTIONAL {?city dbp:populationMetro ?popMetro . }

}
```

The idea of the OPTIONAL clause is to enable to you to bring data *if it exists*, but to ignore it if it does not. This is a key way in which SPARQL deals with sparse data and missing values elegantly.

OPTIONAL operator

- The reason why we need the OPTIONAL operator is because there can be missing information in RDF.
- In the SQL world, missing information is represented with NULL. However, *there are no NULL values in RDF*.
- Either a triple exists or it does not. In fact, since RDF data is independent from its physical data representation, the whole idea of NULL is completely unnecessary.
- NULL values in relational databases are just a manifestation of the tabular logical data model; there needed to be some way to represent an empty cell. In SQL, if in a record a value is NULL and you do SELECT * FROM table, you still get the NULL in the result.
- This is not the case in SPARQL. Instead, you don't get the record at all (as we illustrated in Query 3). If you still want to get the record, you need to use OPTIONAL.

Query 5

This query returns the cities that are of type "Cities in Texas", their total population, and optionally their metro populations. The results are returned in the order of their total populations (so big cities like Houston would be the first results).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT * WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal .
    OPTIONAL {?city dbp:populationMetro ?popMetro . }
}
ORDER BY desc(?popTotal)
```

You can also use asc() option to return the results in ascending order.

Query 6

This query returns the cities that are of type "Cities in Texas", their total population, and optionally their metro populations. The results are returned in the order of their total populations (so big cities would be the top results). At most 10 results will be returned, starting with the 6th result.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT * WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal .
    OPTIONAL {?city dbp:populationMetro ?popMetro . }
}
ORDER BY desc(?popTotal)
LIMIT 10
OFFSET 5
```

Remove results: FILTER

- A FILTER clause restricts which results are returned.
- FILTER operator, uses Boolean conditions to filter out unwanted results. The following filters are allowed:
  - **Logical**: &&, ||, !
  - **Mathematical**: +, -, *, /
  - **Comparison**: =, !=, <, >, <=, >=
  - **SPARQL tests**: isURI, isBlank, isLiteral, bound
  - **SPARQL accessors**: str, lang, datatype
  - **Other**: sameTerm, langMatches, regex

Query 7

This is the same as Query 6, but returns only cities that have a total population of more than 50,000.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT * WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal .
    OPTIONAL {?city dbp:populationMetro ?popMetro . }
    FILTER (?popTotal > 50000)
}
ORDER BY desc(?popTotal)
```

Query 8

This query is the same as query 7, but brings back the human readable name of each city with the results.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT * WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal ;
    rdfs:label ?name
    OPTIONAL {?city dbp:populationMetro ?popMetro . }
    FILTER (?popTotal > 50000)
}
ORDER BY desc(?popTotal)
```

- There are now several result rows for each city resource. Why is that?
- This is a common occurrence in RDF. Unlike SQL, it is very easy to assign multiple values to a resource for a specific property. In this case, there is an rdfs:label for multiple languages in the dataset.

Query 9

Query 8, but requesting *only* English labels for the matching patterns.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT * WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal ;
    rdfs:label ?name
    OPTIONAL {?city dbp:populationMetro ?popMetro. }
    FILTER (?popTotal > 50000 && langmatches(lang(?name), "EN"))
}
ORDER BY desc(?popTotal)
```

The **lang** operator extracts the language tag of the value that is bound to ?name. The **langmatches** operator matches the first language tag with the second language range.

equivalently : FILTER (?popTotal > 50000 **&& lang(?name) = "en"**)

Query 10

This query shows how to use regular expression filters. It is the same as Query 9, but matching only cities with "El" in their names.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT * WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal ;
    rdfs:label ?name
    OPTIONAL {?city dbp:populationMetro ?popMetro. }
    FILTER (?popTotal > 50000 &&
        langmatches(lang(?name), "EN") &&
        regex(str(?name), "El"))
}
ORDER BY desc(?popTotal)
```

Negation

- An important feature in any query language is negation. However, in SPARQL 1.0, there is no explicit negation operator.
- Nevertheless negation is possible through *Negation as Failure* and is written using the OPTIONAL clause, BOUND operator, and the logical not (!) operator. The OPTIONAL operator binds variables to the triples that it wants to exclude, and the filter removes those cases.

Query 11

This query is the same as before, except that it returns *only* cities that *do not* have a metro population.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT * WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal ;
    rdfs:label ?name
    OPTIONAL {?city dbp:populationMetro ?popMetro. }
    FILTER (?popTotal > 50000 && langmatches(lang(?name), "EN") )
    FILTER(!bound(?popMetro))
}
ORDER BY desc(?popTotal)
```

Query 12

This is much the same as the queries that we've been seeing, only it returns cities that are of type "Cities in Texas" *or* of type "Cities in California".

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT * WHERE {
    ?city dbp:populationTotal ?popTotal ;
    rdfs:label ?name
    OPTIONAL {?city dbp:populationMetro ?popMetro. }
    FILTER (?popTotal > 50000 && langmatches(lang(?name), "EN"))
    { ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> . }
    UNION
    { ?city rdf:type <http://dbpedia.org/class/yago/CitiesInCalifornia>. }
}
ORDER BY desc(?popTotal)
```

The UNION clause is a *disjunction* between two basic graph patterns. In other words, it is an OR.

Query 13

This query returns the cities that are of type "Cities in Texas" *and* the graph in which each city resource is contained.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
    GRAPH ?g {
        ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> .
    }
}
```

The results show that all of the data is in the same graph, the default graph.

Query 14

This query asks if Austin is a city in Texas.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ASK WHERE {
    <http://dbpedia.org/resource/Austin,_Texas> rdf:type
        <http://dbpedia.org/class/yago/CitiesInTexas> .
}
```

Query 15

This query returns an RDF graph that describes Austin.

DESCRIBE <http://dbpedia.org/resource/Austin,_Texas>

As mentioned before, the behavior of DESCRIBE is implementation dependent. Virtuoso is the triple store that powers DBpedia, and its implementation of DESCRIBE is to return an RDF graph where the resource is in the subject or in the object position. Other triple stores do not necessary need to have this same behavior.

Query 16

This query returns an RDF graph that describes all the cities in Texas that have a total population greater than 600,000 and a metro population less than 1,800.000.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbp: <http://dbpedia.org/ontology/>
DESCRIBE ?city WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal ;
    dbp:populationMetro ?popMetro.
    FILTER (?popTotal > 600000 && ?popMetro < 1800000)
}
```

Query 17

This query constructs a new RDF graph for cities in Texas that have a metro population greater than 500,000.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
CONSTRUCT {
    ?city rdf:type <http://myvocabulary.com/LargeMetroCitiesInTexas> ;
    <http://myvocabulary.com/cityName> ?name ;
    <http://myvocabulary.com/totalPopulation> ?popTotal ;
    <http://myvocabulary.com/metroPopulation> ?popMetro .
} WHERE {
    ?city rdf:type <http://dbpedia.org/class/yago/CitiesInTexas> ;
    dbp:populationTotal ?popTotal ;
    rdfs:label ?name ;
    dbp:populationMetro ?popMetro .
    FILTER (?popTotal > 500000 && langmatches(lang(?name), "EN"))
}
```

Note that in the CONSTRUCT clause above we have used our own, made up vocabulary!