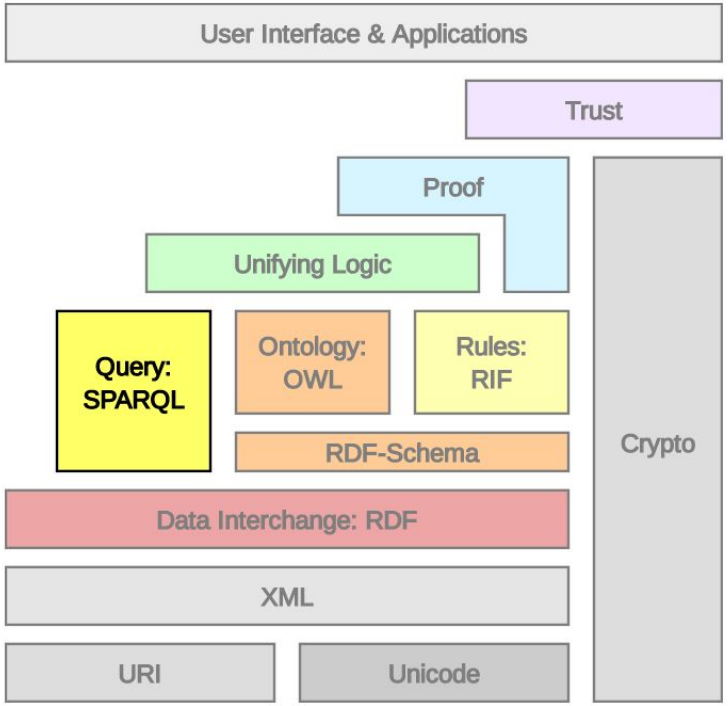
# SPARQL Query Language

# Linked Data Stack - SPARQL



# Outline

- About SPARQL
- Basic SPARQL
  - Presentation
  - Hands-on
- SPARQL in real-life
  - Presentation
  - Hands-on
- Advanced SPARQL
  - Presentation
  - Hands-on

# What is SPARQL?

SPARQL stands for “SPARQL Protocol and RDF Query Language”.
In addition to the language, W3C has also defined:

- The SPARQL Protocol for RDF specification: it defines the remote protocol for issuing SPARQL queries and receiving the results.

- The SPARQL Query Results XML Format specification: it defines an XML document format for representing the results of SPARQL

# Query Languages for RDF and RDFS

There have been many proposals for RDF and RDFS query languages:

- RDQL (http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/)
- ICS-FORTH RQL (http://139.91.183.30:9090/RDF/RQL/) and SeRQL (http://www.openrdf.org/doc/sesame/users/ch06.html)
- SPARQL (http://www.w3.org/TR/rdf-sparql-query/)
- …

In this course we will only cover SPARQL which is the current W3C recommendation for querying RDF data

# SPARQL 1.1

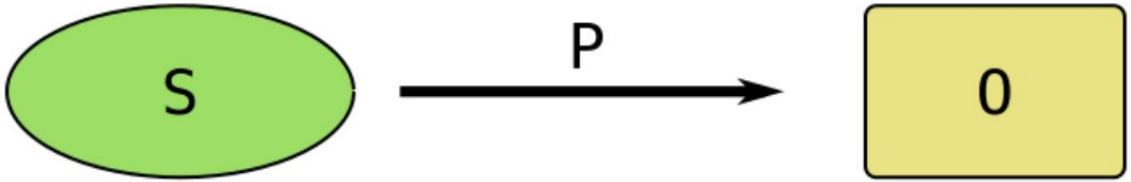
- In this course we will cover most of the SPARQL 1.0 of 2008 and some of SPARQL 1.1.
- The standardization of SPARQL is carried out by the W3C by the SPARQL working group.
- More information about ongoing work by this working group can be found at http://www.w3.org/2009/sparql/wiki/Main_Page
- See http://www.w3.org/TR/sparql11-query/ for the new version of the SPARQL language (SPARQL 1.1).

# Basic SPARQL Structures - Outline

- A bit of RDF and Semantic Web
- First glance at triple patterns
- Components of a SPARQL query
  - Graph patterns
  - Types of queries
  - Modifiers

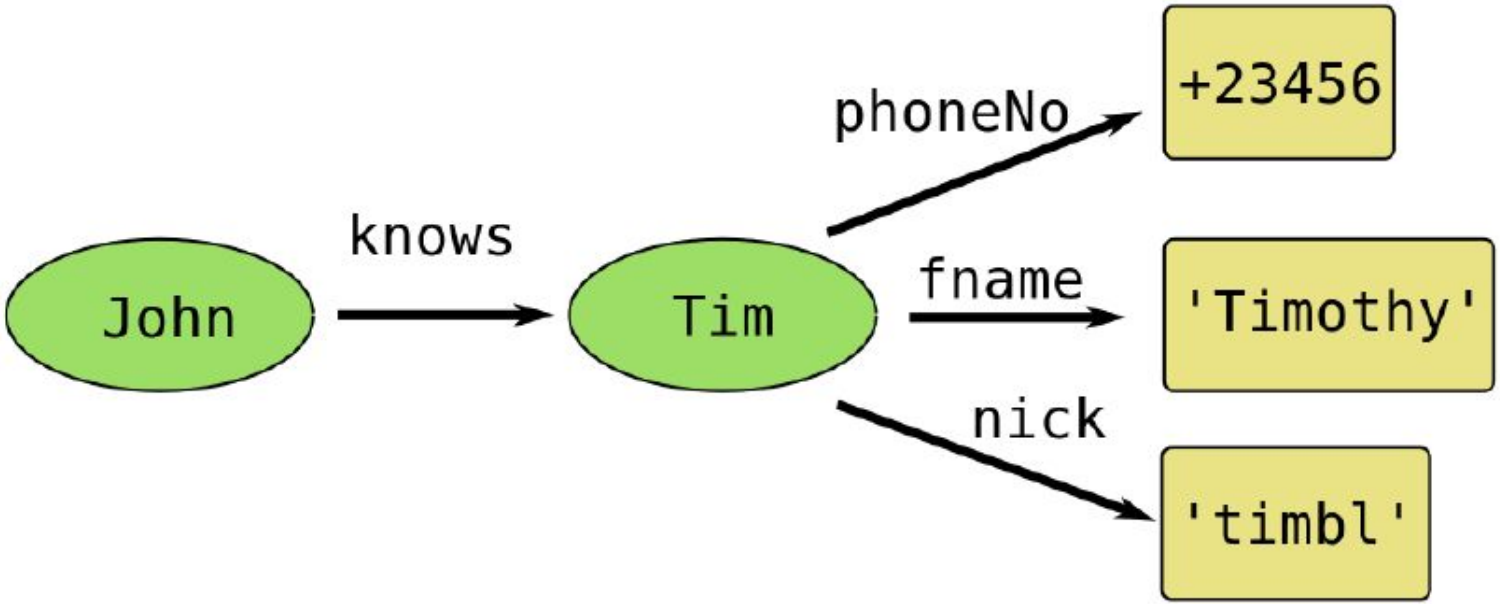
# Triples

Triples are the statements about things (resources), using URIs and Literal values

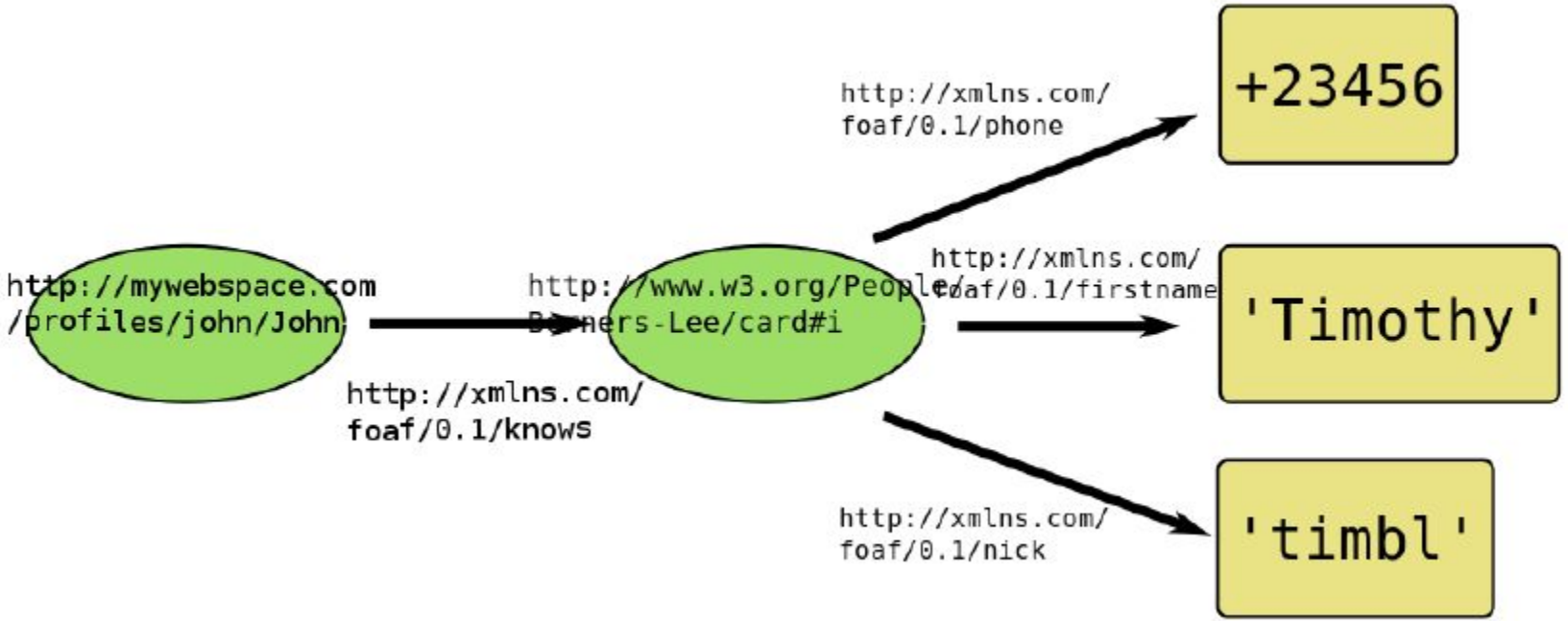


# Triples

# Graph

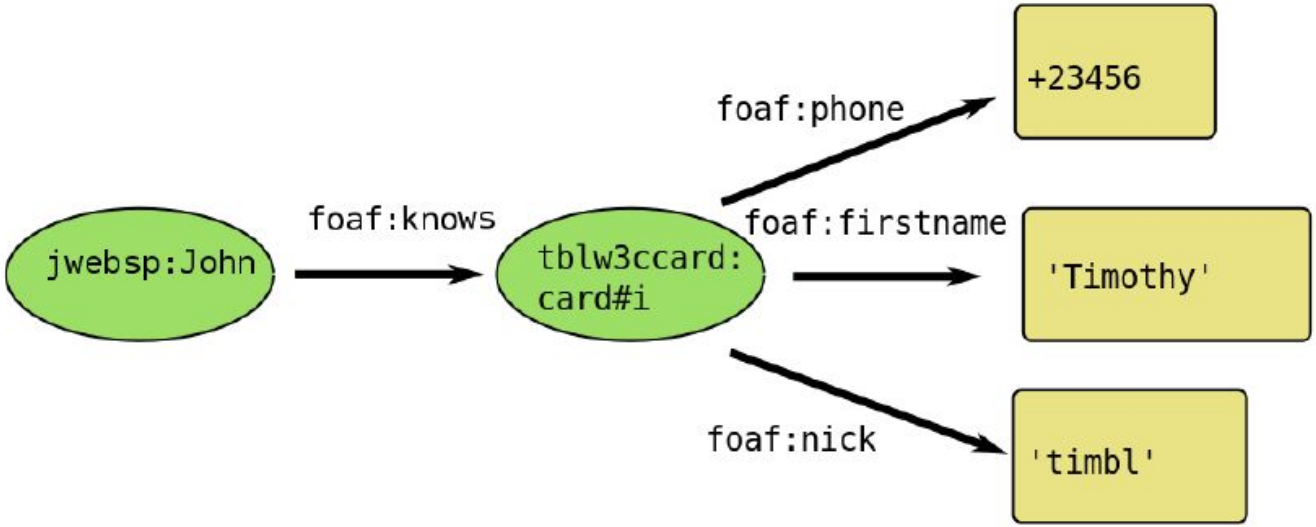


# Graph with URIs



# Prefixes

```
@prefix wbsp: <http://mywebspace.com/profiles/john/>.
@prefix tblw3ccard: <http://www.w3.org/People/Berners-Lee/>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
```



# Vocabularies

- Share concept of a domain
- Utilize URIs as unique identifier
- Define Properties and Classes, and more ....

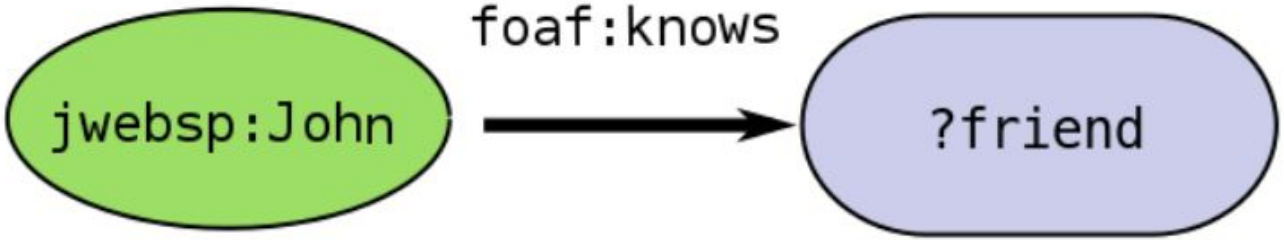
# Well known vocabularies

rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
rdfs : <http://www.w3.org/2000/01/rdf-schema#>
foaf : <http://xmlns.com/foaf/0.1/>
dbpedia : <http://dbpedia.org/resource>

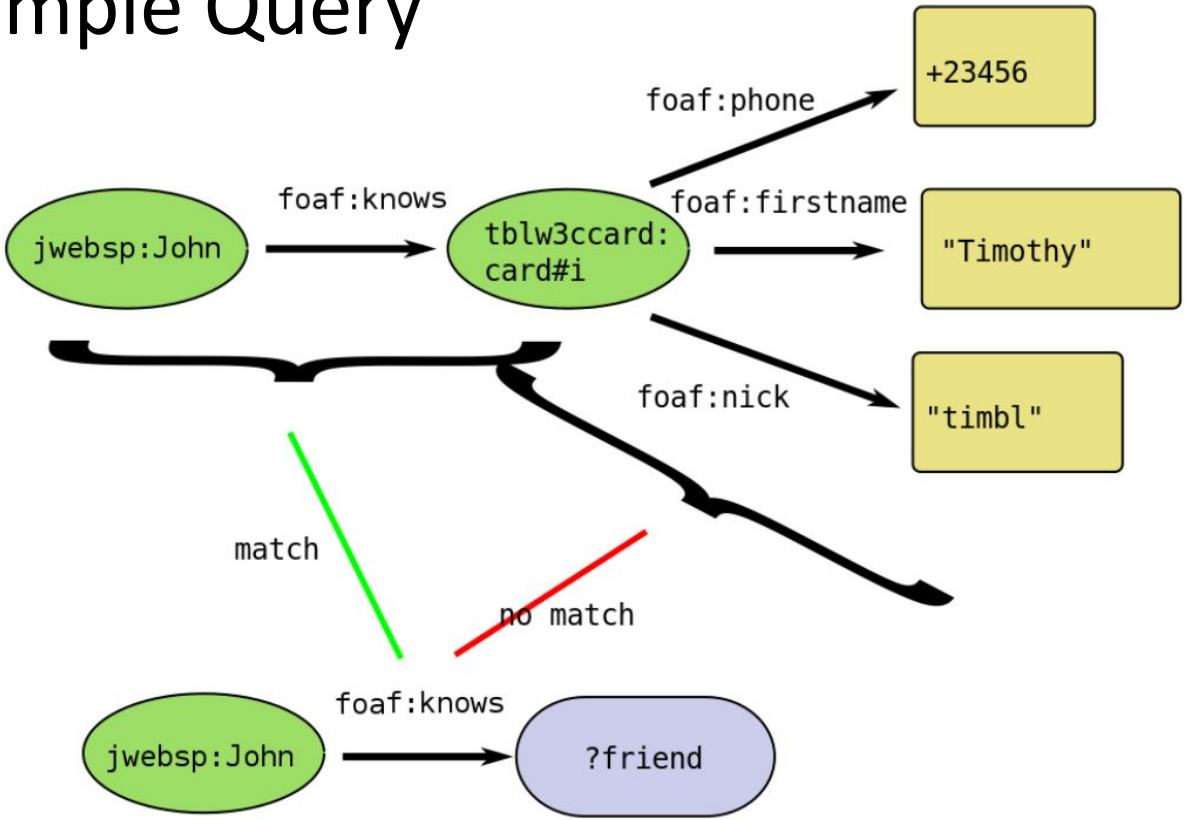
# Triple Stores and SPARQL endpoints

- A SPARQL endpoint exposes one or more Graphs
- HTTP
- expects a parameter "query",
  either with POST or GET with the encoded query
- no required relation between graph name and endpoint name,
  but good practice

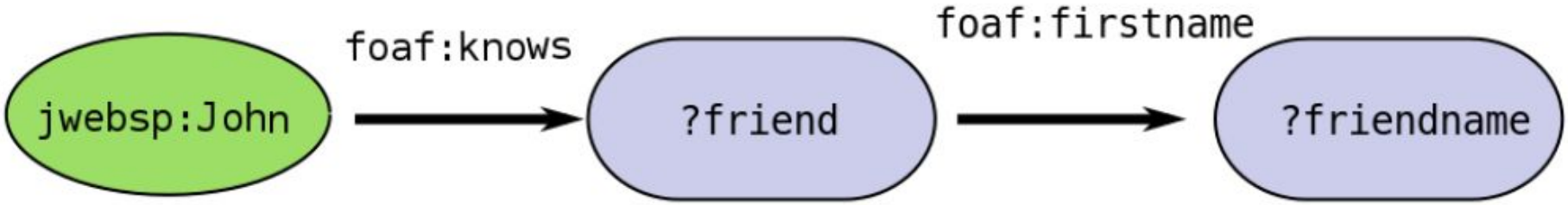
# A simple Query



# A simple Query



# A slightly more complex Query

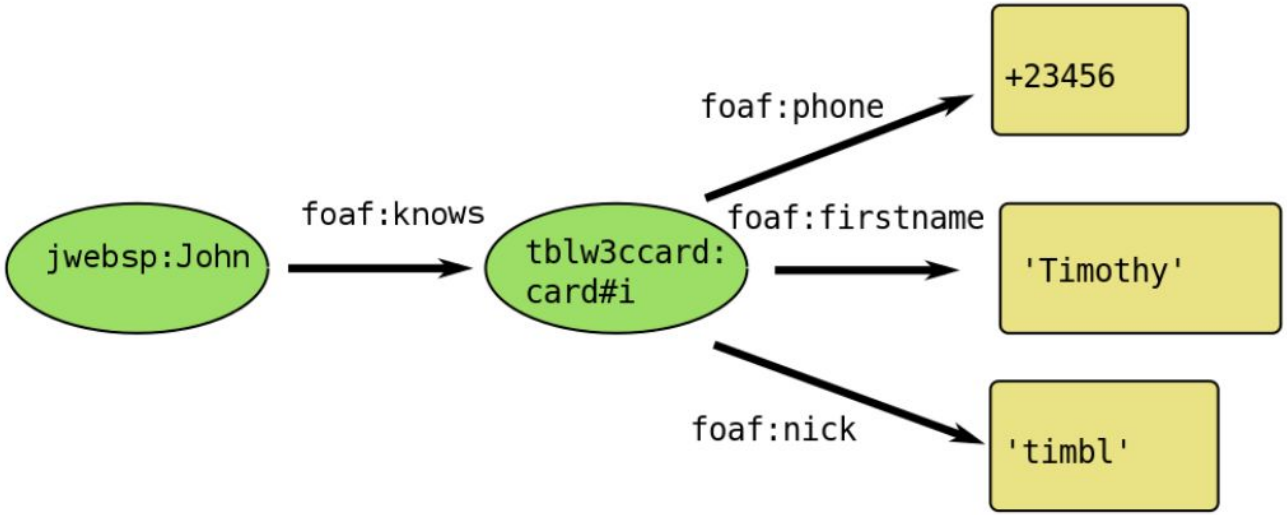


# A slightly more complex Query

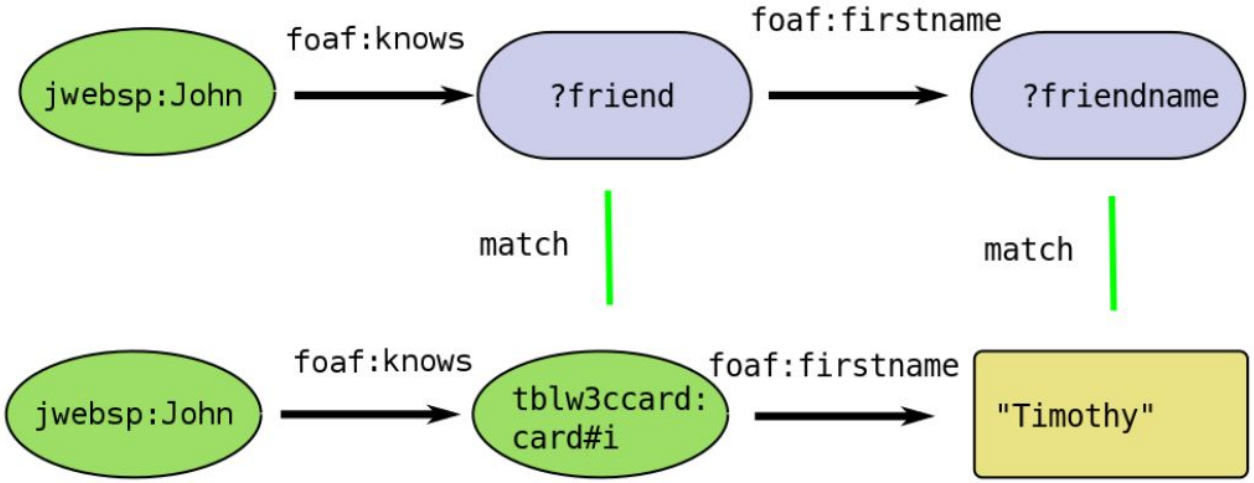
```
SELECT ?friend ?friendname WHERE {
        jwebsp:John foaf:knows ?friend.
        ?friend foaf:firstname ?friendname
}
```

# A slightly more complex Query

```
@prefix wbsp: <http://mywebspace.com/profiles/john/>.
@prefix tblw3ccard: <http://www.w3.org/People/Berners-Lee/>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
```



# A slightly more complex Query




| ?friend | ?friendname |
|---|---|
| tblw3ccard:card#i | "Timothy" |

# Structure of a SPARQL query

```
# prefix declarations
PREFIX ex: <http://example.com/resources/>
....
# query type          # projection          # dataset definition
SELECT                    ?x ?y                    FROM ...

# graph pattern
WHERE {
    ?x a ?y
}

# query modifiers
ORDER BY ?y
```

# Prefixes

Syntactic sugar to keep queries readable

Examples:

PREFIX  :  <http://example.com/base/>

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

<http://xmlns.com/foaf/0.1/knows> == foaf:knows

<http://example.com/base/Tim> == :Tim

# Query Types

| | |
|---|---|
| **SELECT** | returns a result table |
| **ASK** | returns (boolean) true, if the pattern can be matched |
| **CONSTRUCT** | creates triples using templates |
| **DESCRIBE** | returns descriptions of resources |

# From clause

Specifies which graphs should be considered by the endpoint.

- if omitted, the so called default graph is used.
- if specified, the query is evaluated using all specified graphs.
- if specified as named graph, the named graphs can be used in parts of the query.

Graphs can be dereferenced by the SPARQL endpoint.

# Solution modifiers

Change the result of a query

- **LIMIT** and **OFFSET** slice the resultset, useful for pagination
  Example:
  **SELECT** * **WHERE** {.....} **LIMIT** 10
  --> display only 10 results

- **ORDER BY** sorts the result set
  Example:
  **SELECT** * **WHERE** {.....} **ORDER BY ASC**(...) **LIMIT** 10
  --> display the first 10 of the sorted result set

# Where clause

- contains the graph patterns
- conjunctive
- variables are bound to the same values

# Triple patterns

- General form a triple (s p o)
- On all positions variables may occur
- Variables are bound by the SPARQL endpoint

# Triple patterns - Example

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

SELECT ?name WHERE {
        :John foaf:name ?name
}
-->     "John"
```

# Triple patterns - Example

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

SELECT ?friend WHERE {
        :John foaf:knows ?friend
}
-->     :Tim
```

# Triple patterns - Example

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

SELECT ?friend ?name WHERE {
        :John foaf:knows ?friend.
        :John foaf:name ?name.
}
-->     :Tim "John"
```

# Triple patterns - Example

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

SELECT ?friendsname WHERE {
        :John foaf:knows ?friend.
        ?friend foaf:name ?friendsname.
}
-->    "Tim"
```

# Triple patterns - Cartesian product

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .
```

```
SELECT ?person ?friendsname WHERE {
        ?person foaf:knows ?friend.
        ?somebody foaf:name ?friendsname.
}
```

```
:John "John"
:John "Tim"
:Tim "John"
:Tim "Tim"
```

# Matching Resources

Match character by character

either with prefix or full <URI>

- foaf:name == <http://xmlns.com/foaf/spec/name>

percent encoding of reserved characters (like space)

- myns:John%20Doe != myns:John Doe | **Error!**

**case sensitive**

- foaf:name != <http://xmlns.com/foaf/spec/Name>

# Matching Literals

Literals need to match for equality character-by-character

- can have datatype: xsd:int, xsd:date
  - SPARQL engine may know interpretation of datatype
  - for equality, it needs to match exactly
- can have language tag

# Filter

- operate on graph patterns
- testing values
- most prominently: restrict Literal values
  - string comparison
  - regular expressions
  - numeric comparators
- type/language checks
- evaluate in the end either to true, false or type error

# Filter Overview

- Logical: !, &&, ||
- Math: +, -, *, /
- Comparison: =, !=, >, <, ...
- SPARQL tests: isURI, isBlank, isLiteral, bound
- SPARQL accessors: str, lang, datatype
  Other: sameTerm, langMatches, regex
- Vendor specific: prefixed like bif:contains

# String Filtering

- str() just the literal value, without datatype
- regex() full regular expression
- bif:contains string search using special index

# String Filtering Example

```
:John :age 32 .
:John foaf:name "John"@en .
:Tim :age 20.
:Tim foaf:name "Tim"^^xsd:string .
```

```
SELECT ?friend {
        ?friend foaf:name "Tim".
}
```

--> empty

# String Filtering Example

```
:John :age 32 .
:John foaf:name "John"@en .
:Tim :age 20.
:Tim foaf:name "Tim"^^xsd:string .
```

```
SELECT ?friend {
        ?friend foaf:name ?name.
        FILTER (str(?name) = "Tim")
}
```

--> :Tim

# String Filtering Example

```
:John :age 32 .
:John foaf:name "John"@en .
:Tim :age 20.
:Tim foaf:name "Tim"^^xsd:string .
```

```
SELECT ?friend {
        ?friend foaf:name ?name.
        ?name bif:contains "im".
}
```

--> :Tim

# Language and Datatype Filtering

- lang(?x)  accessor to the language of a literal
- langMatches(lang(?x),"en") evaluates
  if a language tag matches another language tag
- datatype(?x) accesses the datatype of the literal ?x

# Numeric Filtering

```
:John :age 32 .
:John foaf:name "John"@en .
:Tim :age 20.
:Tim foaf:name "Tim"^^xsd:string .

SELECT ?friend WHERE {
        ?friend :age ?age .
        FILTER (?age>25)
}
-->   :John
```

# Logical Operators

```
:John :age 32 .
:John foaf:name "John"@en .
:Tim :age 20.
:Tim foaf:name "Tim"^^xsd:string .

SELECT ?friend WHERE {
        ?friend foaf:name ?name.
        ?friend :age ?age .
        FILTER ( str(?name) = "Tim" && ?age>25 )
}
-->   NULL
```

Note: str(?name) = "Tim"  has single equality sign !

# Logical Operators

```
:John :age 32 .
:John foaf:name "John"@en .
:Tim :age 20.
:Tim foaf:name "Tim"^^xsd:string .

SELECT ?friend WHERE {
        ?friend foaf:name ?name.
        ?friend :age ?age .
        FILTER ( str(?name) = "Tim" || ?age>25 )
}
-->   :Tim
-->   :John
```

# Optional values

- Similar to left join in SQL
- Allows querying for incomplete data
- “Optional” takes a full graph pattern
- Syntax {pattern1} OPTIONAL {optpattern}

# Optional Example

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:John foaf:phone "+123456" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

SELECT ?name ?phone WHERE {
        ?person foaf:name ?name.
        ?person foaf:phone ?phone.
}
--> "John" "+123456"
```

This is a bit unsatisfying

# Optional Example

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:John foaf:phone "+123456" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

SELECT ?name ?phone WHERE {
        ?person foaf:name ?name.
        OPTIONAL {?person foaf:phone ?phone . }
}
--> "John" "+123456"
--> "Tim"
```

# Union

Syntax: {graph pattern} **UNION** {graph pattern}

Allows querying (partly) differing data structures

# Union Example

```
:John rdf:type foaf:Person .
:John foaf:name "John" .
:Tim rdf:type foaf:Person .
:Tim foaf:name "Tim" .
:Jane rdf:type foaf:Person .
:Jane rdfs:label "Jane" .
```

```
SELECT ?name WHERE {
        ?person a foaf:Person.
        ?person foaf:name ?name
}
--> "John"
--> "Tim"
```

# Union Example

```
:John rdf:type     foaf:Person .
:John foaf:name "John" .
:Tim   rdf:type     foaf:Person .
:Tim   foaf:name "Tim" .
:Jane rdf:type     foaf:Person .
:Jane rdfs:label  "Jane" .
```

```
SELECT ?name WHERE {
        ?person a foaf:Person.
        {?person foaf:name ?name} UNION {?person rdfs:label ?name}
}
--> "John"
--> "Tim"
--> "Jane"
```

# Union Example

```
:John rdf:type     foaf:Person .
:John foaf:name "John" .
:Tim   rdf:type     foaf:Person .
:Tim   foaf:name "Tim" .
:Jane rdf:type     foaf:Person .
:Jane rdfs:label  "Jane" .
```

```
SELECT ?name WHERE {
      {?person foaf:name ?name. ?person a foaf:Person} UNION
      {?person rdfs:label ?name. ?person a foaf:Person. }
}
--> "John"
--> "Tim"
--> "Jane"
```

# Projection

**SELECT** * **WHERE** {.....}

→ all variables mentioned in the graph patterns

**SELECT** ?s ?o **WHERE** {?s ?p ?o}

→ only the variables specified, in this case ?s and ?o

**SELECT DISTINCT** .....

→ eliminates duplicates in the result

# Count

a simple aggregate function
counts how often a variable is bound.

Example:

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .
```

```
SELECT count(?person) WHERE {
        ?person foaf:name ?name .
}
```

→ 2

# SPARQL in Real-Life - Outline

- We use the previously acquired knowledge for
  - Exploring unknown data structures and vocabularies
  - Querying inconsistent data structures

# Some public SPARQL endpoints

SPARQLer: general-purpose query endpoint for Web-accessible data

DBpedia: extensive RDF data from Wikipedia

DBLP: bibliographic data from computer science journals and conferences

LMDB: data from MDB - Movies database (without html form)

World Factbook: country statistics from the CIA World factbook

# About DBpedia

- Crystallization point of the Semantic Web
- Single most important data source
- Community effort
- Extract from the semi-structured information in Wikipedia
- Non-curated content

# Know your limits!

The DBpedia endpoint popular and well-used

Always add a LIMIT statement, when constructing queries

# Vocabulary Exploration

Exploration by examining instance data

- Find descriptive information about the dataset
- Use tools
- Analyze the query dump
- Dereference URI
- Queries

# Descriptive Information

Most datasets have publications describing them

Find papers about them using scholar.google.com

# Tools: Relationship finder

http://www.visualdataweb.org/relfinder.php

# Dereference URIs

The Linked Data principles allow dereferencing URIs to get descriptions

Instance data on Leipzig

--> http://dbpedia.org/resource/Leipzig

Vocabulary information about foaf:name

--> http://xmlns.com/foaf/0.1/name

# Querying

```
DESCRIBE <http://dbpedia.org/resource/Leipzig>

SELECT ?p ?o WHERE {
        <http://dbpedia.org/resource/Leipzig> ?p ?o .
}
```

# ?p queries

Query resources with a variable in the predicate position of a triple pattern

# ?p queries - Example

```
:John foaf:name "John".
:John rdfs:label "This is John".
:John foaf:phone "+12312".

SELECT  ?p ?o WHERE {
      :John ?p ?o .
}
-->  foaf:name     "John"
-->  rdfs:label    "This is John"
-->  foaf:phone    "+12312"
```

# Querying for Classes

Vocabularies define classes

- foaf:Person
- foaf:Document

rdf:type/a associates **an instance** with **a class**

- :John a foaf:Person == :John rdf:type foaf:Person

# Querying for Classes - Example

:John a foaf:Person .
:Pluto a animals:Dog .

**SELECT** ?person **WHERE** {
    ?person a foaf:Person .
}
→ :John

**SELECT** ?class **WHERE** {
    ?instance a ?class .
}
→ foaf:Person
→ animals:Dog

# Some public SPARQL endpoints

SPARQLer: general-purpose query endpoint for Web-accessible data

DBpedia: extensive RDF data from Wikipedia

DBLP: bibliographic data from computer science journals and conferences

LMDB: data from MDB - Movies database (without html form)

World Factbook: country statistics from the CIA World factbook

# Types

Get all the possible types of concepts in DBpedia

# Types A

```
SELECT distinct ?type

WHERE {
   ?e a ?type .
}
```

# Properties list

Get all the properties of the Actor class. Show also their titles

# Properties list A

```
SELECT distinct ?p ?title

WHERE {
   ?p rdfs:label ?title .
   ?e a <http://dbpedia.org/ontology/Actor> .
   ?e ?p ?v .
}
```

# Working with DBpedia page

- Look through Ivan The Terrible DBpedia page. What properties you might use to get the full list of Russian Leaders?
- Check your suggestions using the DBpedia endpoint
- Compare the amount of results for different queries using COUNT aggregation function.

# Working with DBpedia page A

```
SELECT ?e WHERE {
   ?e dcterms:subject category:Russian_leaders .
}

SELECT ?e WHERE {
   ?e dbpprop:title dbpedia:List_of_Russian_rulers .
}
...
SELECT count(?e) WHERE {
   ?e dbpprop:title dbpedia:List_of_Russian_rulers .
}

...
```

# Multiple patterns

Change the previous query to show also the real name of the leader.

# Multiple patterns A

```
SELECT ?e ?name WHERE {
   ?e dbpprop:title dbpedia:List_of_Russian_rulers  .
   ?e dbpprop:name ?name .
}
```

Better version:

```
SELECT ?e ?name WHERE {
   ?e dbpprop:title dbpedia:List_of_Russian_rulers  .
   ?e rdfs:label ?name .
}
```

# LIMIT

Show only 20 first results. Then show the next 20.

Show twenty results starting from the 10th.

# LIMIT A

```
SELECT ?e ?name WHERE {
   ?e dbpprop:title dbpedia:List_of_Russian_rulers  .
   ?e rdfs:label ?name .
}
LIMIT 20
OFFSET 10
```

# FILTER

Filter the list and show only the results for Ivan_the_Terrible

# FILTER A

```
SELECT ?e ?name WHERE {
   ?e dbpprop:title dbpedia:List_of_Russian_rulers  .
   ?e rdfs:label ?name .

FILTER (?e = <http://dbpedia.org/resource/Ivan_the_Terrible>)
}
```

# String Matching

Show the list of all Russian leaders with the name "Ivan"

# String matching A

```
SELECT ?e ?name WHERE{
   ?e dbpprop:title dbpedia:List_of_Russian_rulers  .
   ?e rdfs:label ?name .

FILTER regex(?name, "ivan", "i")
}
```

# Langmatching

Get a list of Russian leaders showing only Russian labels for the name.

# Langmatching A

```
SELECT ?e ?name WHERE {
   ?e dbpprop:title dbpedia:List_of_Russian_rulers  .
   ?e rdfs:label ?name .

FILTER ( langMatches( lang(?name), "RU" ) )
}
```

# Choosing properties to show

Rewrite the previous query to show :

- the entry,
- the name,
- the name of predecessor and
- the name of successor.

# Choosing properties to show A

```
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpedia:<http://dbpedia.org/resource/>
PREFIX dbo:<http://dbpedia.org/ontology/>

SELECT  ?e  ?name  ?predecessor_name ?successor_name WHERE {
        ?e dbpprop:title dbpedia:List_of_Russian_rulers .
        ?e rdfs:label ?name .
        ?e dbo:successor ?successor .
        ?successor rdfs:label ?successor_name .
        ?e dbo:predecessor ?predecessor .
        ?predecessor rdfs:label ?predecessor_name .
        FILTER (    langMatches(lang(?name), "EN") &&
                    langMatches(lang(?successor_name), "EN") &&
                    langMatches(lang(?predecessor_name), "EN")
                )
}
```

# More practice

Find a real name of the Russian leader,
who was on the throne right before Catherine I
("Catherine I of Russia"@en)

Can  you find other ways to do the same task?

# More practice A

```
SELECT ?name WHERE{
   ?e dbpprop:title dbpedia:List_of_Russian_rulers .
   ?e rdfs:label ?name .
   ?e dbpedia-owl:successor ?successor  .
   ?successor rdfs:label "Catherine I of Russia"@en
}

SELECT ?name as ?leader WHERE {
   ?e dbpprop:title dbpedia:List_of_Russian_rulers .
   ?e rdfs:label ?name .
   ?e dbpedia-owl:successor ?successor .
   ?successor  rdfs:label ?successor_name .
   FILTER (?successor_name = "Catherine I of Russia"@en)
}
```

# OPTIONALs

Look at the page: http://dbpedia.org/page/Dmitry_of_Suzdal

Why this leader is not in the results of the previous queries?

Fix the problem.

# OPTIONALs A

```
SELECT ?e ?name ?predecessor_name ?successor_name WHERE {
        ?e dbpprop:title dbpedia:List_of_Russian_rulers .
        ?e rdfs:label ?name .
        FILTER ( langMatches( lang(?name), "EN" ) ) .
        OPTIONAL {
                ?e dbpedia-owl:successor ?successor .
                ?successor rdfs:label ?successor_name .
                FILTER ( langMatches( lang(?successor_name), "EN" ) )  .
         }  .
        OPTIONAL {
                ?e dbpedia-owl:predecessor ?predecessor .
                ?predecessor rdfs:label ?predecessor_name .
                FILTER ( langMatches( lang(?predecessor_name), "EN" ) ) .
   }
}
```

# UNIONs

Look at the http://dbpedia.org/page/Dmitry_of_Suzdal page more carefully. What can you say about successor and predecessor of the leader?

Fix the problem.

# UNIONs A

```
SELECT ?e ?name ?predecessor_name ?successor_name WHERE {
        ?e dbpprop:title dbpedia:List_of_Russian_rulers .
        ?e rdfs:label ?name .
        FILTER ( langMatches( lang(?name), "EN" ) ) .
        OPTIONAL {
                {?e dbpedia-owl:successor ?successor}  UNION  { ?e dbpprop:after ?successor } .
                ?successor rdfs:label ?successor_name .
                FILTER ( langMatches( lang(?successor_name), "EN" ) )  .
         }  .
         OPTIONAL {
                {?e dbpprop:predecessor ?predecessor} UNION  { ?e dbpprop:before ?predecessor } .
                ?predecessor rdfs:label ?predecessor_name .
                FILTER ( langMatches( lang(?predecessor_name), "EN" ) ) .
        }  .
}
```

# Final task

Show the list of actors, played together with Julia Roberts.
For each result show also the name of the movie and the director.
Order the results both by director and by movie.

# Final task A

```
SELECT ?director_name ?movie_name ?actor_name WHERE {
        ?movie dbpedia-owl:starring <http://dbpedia.org/resource/Julia_Roberts> .
        ?movie dbpedia-owl:starring ?actor .
        ?movie rdfs:label ?movie_name .
        ?actor rdfs:label ?actor_name .
        ?movie dbpedia-owl:director ?director .
        ?director rdfs:label ?director_name .
        FILTER ( langMatches( lang(?movie_name), "EN") &&
                 langMatches( lang(?actor_name), "EN") &&
                 langMatches( lang(?director_name), "EN") ) .
 }
ORDER BY ?director ?movie
```

# Aggregate Functions

Aggregate functions similar to SQL were introduced with SPARQL 1.1

Most important min, max, avg, sum, count

Group by groups the results accordingly, necessary for projection

# Aggregate Functions - Example

```
:John :age 32 .
:John :gender :male .
:Tim :age 20.
:Tim :gender :male.
:Jane :gender :female.
:Jane :age 23.

SELECT avg(?age) WHERE {?person :age ?age}
-->     25

SELECT ?gender min(?age)  WHERE {
        ?person :age ?age.
        ?person :gender ?gender.
}  GROUP BY ?gender

-->     :male  20
-->     :female 23
```

# Other query types

**CONSTRUCT**
→ creates a graph by binding variables in a template

**ASK**
→ returns a boolean values, if the pattern could be found

**DESCRIBE**
→ gives a short description about some resources

# Other Query Types Examples

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:John foaf:phone "+123456" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

CONSTRUCT {
        ?person foaf:name ?name.
        ?person foaf:phone ?phone.
}  WHERE {
        ?person foaf:name ?name.
        ?person foaf:phone ?phone.
}
--> :John foaf:name "John" .
--> :John foaf:phone "+123456" .
```

# Other Query Types Examples

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:John foaf:phone "+123456" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .
```

```
DESCRIBE ?person WHERE {
        ?person foaf:name ?name.
        ?person foaf:phone ?phone
}
```

```
--> :John foaf:name "John" .
--> :John foaf:phone "+123456" .
```

# Other Query Types Examples

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:John foaf:phone "+123456" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .
```

```
ASK  {
        ?person foaf:name ?name.
        ?person foaf:phone ?phone.
}
```

→ true

# Named Graphs

Allow more control about from which graphs a triple is coming from

```
SELECT *  FROM NAMED <http://mygraph.example/>
WHERE{
      GRAPH ?g {?s ?p ?o}
}
```

→ <http://mygraph.example/> <s> <p> <o>

# Named Graphs - Example

```
SELECT ?g ?o ?p2 ?o2
FROM <http://www.w3.org/People/Berners-Lee/card.rdf>
FROM NAMED <http://dig.csail.mit.edu/2008/webdav/timbl/foaf.rdf>
{
      ?s ?p ?o.
      GRAPH ?g {?o ?p2 ?o2}
}
```

→ A huge list of triples

# Subquery with from clause

Subqueries on the cheap:

1. Write the query which you want to use as basis as a CONSTRUCT Query
2. URL encode it
3. Create the other query
4. Put the endpoint for the first + the encoded query in the FROM clause of the other query.

→ SELECT * FROM <http://dbpedia.org/sparql?query=SELECT%20....>

# Negation

Question: How to find all contacts that do NOT have a phone number

Use a combination of not, bound and optional!

# Negation Example

```
:John foaf:knows :Tim .
:John foaf:name "John" .
:John foaf:phone "+123456" .
:Tim foaf:knows :John .
:Tim foaf:name "Tim" .

SELECT ?name ?phone WHERE {
        ?person foaf:name ?name.
        OPTIONAL {?person foaf:phone ?phone}
        FILTER ( !bound(?phone) )
}
```

→ Tim

# Query Federation

The SERVICE keyword allows federation between multiple SPARQL endpoints

The endpoints distribute the query

# Reasoning and SPARQL

Reasoning is not a SPARQL feature

Some reasoning can be simulated with SPARQL

Missing direct associations with parent classes can be queried with patterns like

{?sub rdfs:subClassof ?parent .
?subsub rdfs:subClassOf ?sub...}

# Property Paths

Either syntactic sugar:

```
?person foaf:knows/foaf:name ?name ==
?person  foaf:knows ?friend. ?friend foaf:name ?name
```

Or explorative:

```
?x foaf:knows+/foaf:name ?name .
```

The SPARQL 1.1 Recommendation has further helpful examples.

# Queries and Algebra

- SPARQL queries are compiled into algebraic expressions for evaluation
- SPARQL queries with identical result sets can perform differently, depending on how well the query can be optimized.

**Examples**:

```
select * {?s ?p ?o. FILTER (?p = foaf:name && ?o = "Angela Merkel"@en) }
```

```
select * {?s foaf:name "Angela Merkel"@en }
```

**Note** : Valid SPARQL queries (small letters, no indenting,  and where clause missing )

# Algebra

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

SELECT  * WHERE  { ?s foaf:name "Angela Merkel"@en }

compiles into

```
1 (base <http://example/base/>
2   (prefix ((foaf: <http://xmlns.com/foaf/spec/>))
3     (bgp (triple ?s foaf:name "Angela Merkel"@en))))
```

# Algebra

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

SELECT  * WHERE   {
        ?s ?p ?o .
       FILTER ( ( ?p = foaf:name ) && ( ?o = "Angela Merkel"@en ) )
 }
```

compiles into

```
(base <http://example/base/>
  (prefix ((foaf: <http://xmlns.com/foaf/spec/>))
    (filter (&& (= ?p foaf:name) (= ?o "Angela Merkel"@en))
      (bgp (triple ?s ?p ?o)))))
```

# Evaluation

- SPARQL queries are recursively evaluated, starting from the triple patterns (leaf nodes)
- Intermediate result sets are build

Usage of indexes for:

- Resources
- Literals

But not for:

- regex
- aggregate functions

Instead consider bif:contains (bif = built-in function, in some engines)
Also consider pushing filters as deep into queries as possible.

# Not Bound

Check your final task from the basic SPARQL tutorial:

Try to find the movies, starring by Julia Roberts,
where there is no information about director.

# Not Bound A

```
SELECT ?movie_label WHERE {
        ?movie dbpedia-owl:starring <http://dbpedia.org/resource/Julia_Roberts> .
        ?movie rdfs:label ?movie_label .
        OPTIONAL {?movie dbpedia-owl:director ?director} .
        FILTER (langMatches(lang(?movie_label), "EN") && !bound(?director) )
 }
```

# Aggregation

Collect the statistics about Russia:

Find the population, total number of cities,
number of cities with the population more than 1 billion,
average population of cities.

# Aggregation A1

```
SELECT ?population count(?city) WHERE {
        <http://dbpedia.org/resource/Russia> dbpprop:populationEstimate ?population .
        ?city a dbpedia-owl:PopulatedPlace .
        ?city dbpedia-owl:country <http://dbpedia.org/resource/Russia> .
}
```

# Aggregation A2

```
SELECT count(?billioners) WHERE {
        ?billioners a dbpedia-owl:PopulatedPlace .
        ?billioners dbpedia-owl:country <http://dbpedia.org/resource/Russia> .
        ?billioners dbpprop:pop2002census ?city_population .
        FILTER (?city_population > 1000000)
}
```

# Aggregation A3

```
SELECT AVG(?population) WHERE {
        ?city a dbpedia-owl:PopulatedPlace .
        ?city dbpedia-owl:country <http://dbpedia.org/resource/Russia> .
        ?city dbpprop:pop2002census ?population .
}
```

# AS

Change the previous queries to show the correct titles of the table columns.

# AS A

```
SELECT ?population count(?city) AS ?number_of_cities WHERE {
        <http://dbpedia.org/resource/Russia> dbpprop:populationEstimate ?population .
        ?city a dbpedia-owl:PopulatedPlace .
        ?city dbpedia-owl:country <http://dbpedia.org/resource/Russia> .
}
```

# MINUS

Exclude the Novosibirsk when counting the average population of Russian cities

# MINUS A

```
SELECT AVG(?population) WHERE {
        ?city a dbpedia-owl:PopulatedPlace .
        ?city dbpedia-owl:country <http://dbpedia.org/resource/Russia> .
        ?city dbpprop:pop2002census ?population .
         MINUS
        {<http://dbpedia.org/resource/Novosibirsk> dbpprop:pop2002census ?population}
}
```

# Retrieving the information

Show the information about Moscow.
Show all triples, where Moscow is either a subject or an object.

# Retrieving the information A

```
SELECT ?s ?p ?o WHERE {
    { ?s ?p ?o. filter (?s = <http://dbpedia.org/resource/Moscow>) }
    UNION
    { ?s ?p ?o. filter (?o = <http://dbpedia.org/resource/Moscow>) }
}
```

# Searching for commons

Find the commons between Mikhail Gorbachev and Ivan The Terrible.

# Searching for commons A

```
SELECT ?p ?c ?o WHERE {
        <http://dbpedia.org/resource/Ivan_the_Terrible> ?p ?o .
        <http://dbpedia.org/resource/Mikhail_Gorbachev> ?c ?o
}
```

# Use of relational finder

Do the same task in RelFinder

# Use of Hanne

Find the best way to get the list of Russian football clubs using Hanne

# How to connect

Download the file

# SPARQL Endpoints

http://www.sparql.org/

http://dbpedia.org/sparql

http://www.w3.org/wiki/SparqlEndpoints

# SPARQL enabled triple stores

Virtuoso Open Source --- http://virtuoso.openlinksw.com/

Jena & Fuseki --- http://jena.apache.org/

Sesame --- http://www.openrdf.org/

# Local queries with ARQ

How to query RDF datasets in local files:

1. Download Jena
2. Learn about the SPARQL features that ARQ supports
3. Use the arq.query command-line tool ( documentation )
4. java -cp <jena>/lib/commons-codec-1.6.jar:…:<jena>/lib/xml-apis-1.4.01.jar arq.query --data=file.rdf --query=file.sparql
   - Wrap this into a shell script or alias to save time!
   - Java class path must contain all *.jar files of Jena
   - on Windows use ; instead of : as separator
   - data file must be RDF/XML and have *.rdf filename extension
     - Most *.owl files are RDF/XML
     - use Jena's rdfcat (“jena.rdfcat” instead of “arq query --data…” in the above command line) or Protégé to convert other RDF serializations
   - trick in Unix-style shells (e.g. bash): instead of --query=file.sparql use --query=<(echo "SELECT ...") (“process substitution”)

# ARQ: Output

Example of running ARQ (see previous slide for full command line):

$ ... arq.query --data=test.rdf --query=<(echo "SELECT DISTINCT ?class WHERE {?s a ?class}")

```
-----------------------------------------
| class                                 |
=========================================
| <http://xmlns.com/foaf/0.1/Person>    |
| <http://xmlns.com/foaf/0.1/Document>  |
-----------------------------------------
```

# Additional Tools

YASGUI – a user-friendly web GUI to query a given SPARQL endpoint, with syntax highlighting.

FedX --- http://www.fluidops.com/fedx/

# Further Learning Resources

- SPARQL Trainer (http://aksw.org/projects/sparqltrainer)
- Learning SPARQL, Bob DuCharme, O'Reilly (2011)
- Semantic Web for the Working Ontologist,
  Dean Allemang and James Hendler, Morgan Kaufmann (2011)
- SPARQL by example,
  http://www.cambridgesemantics.com/semantic-university/sparql-by-example

# Additional Topics

GeoSparql

Task: Display monuments 30km away on a map.

Sparql Update

Task: Create a Graph with some personal information about you.

# The end!

# How to install fuseki and use it

## Getting Started With Fuseki

This section provides a brief guide to getting up and running with a simple server installation. It uses the SOH (SPARQL over HTTP) scripts included in the download.

1. Download (this includes the server and the SOH scripts)
2. Unzip
3. (Linux) `chmod +x fuseki-server s-*`
4. Run a server

   fuseki-server --update --mem /ds

The server logging goes to the console:

```
09:25:41 INFO  Fuseki               :: Dataset: in-memory
09:25:41 INFO  Fuseki               :: Update enabled
09:25:41 INFO  Fuseki               :: Fuseki development
09:25:41 INFO  Fuseki               :: Jetty 7.2.1.v20101111
09:25:41 INFO  Fuseki               :: Dataset = /ds
09:25:41 INFO  Fuseki               :: Started 2011/01/06 09:25:41 GMT on port 3030
```

Source: http://jena.apache.org/documentation/serving_data/

# Define your own knowledge base and load it to fuseki and query it

**This is the knowledge base**

John rdf:type foaf:Person .
John foaf:name "John" .
Tim rdf:type foaf:Person .
Tim foaf:name "Tim" .
Jane rdf:type foaf:Person .
Jane rdfs:label "Jane" .
Jane foaf:name "Jane" .

**and this is the query:**

```
SELECT ?name WHERE {
        ?person a foaf:Person.
        {?person foaf:name ?name} UNION {?person rdfs:label ?name}
}
```

**You can also try more queries inspiring from the lecture.**