Building and Mining Knowledge graphs

(KEN4256)

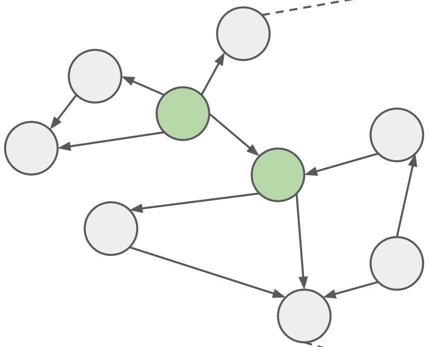
Lecture 4: KG Retrieval (SPARQL)

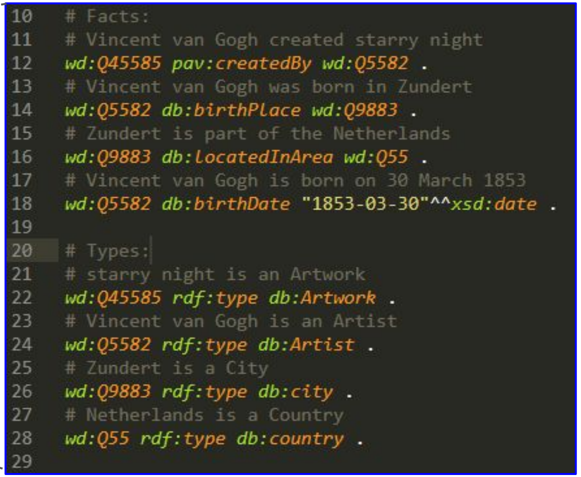
Maastricht University

Institute of Data Science

id: KEN4256_L4
version: 1.2024.0
created: February 2, 2019
last modified: March 26, 2024
published on: March 26, 2024

Today





```
# Facts:
# Vincent van Gogh created starry night
wd:Q45585 pav:createdBy wd:Q5582 .
# Vincent van Gogh was born in Zundert
wd:Q5582 db:birthPlace wd:Q9883 .
# Zundert is part of the Netherlands
wd:Q9883 db:locatedInArea wd:Q55 .
# Vincent van Gogh is born on 30 March 1853
wd:Q5582 db:birthDate "1853-03-30"^^xsd:date .

# Types:
# starry night is an Artwork
wd:Q45585 rdf:type db:Artwork .
# Vincent van Gogh is an Artist
wd:Q5582 rdf:type db:Artist .
# Zundert is a City
wd:Q9883 rdf:type db:city .
# Netherlands is a Country
wd:Q55 rdf:type db:country .
```

SPARQL

?

Where was Vincent van Gogh born?

I already have a Knowledge Graph, how can I retrieve relevant information from it?

SPARQL W3C specification

- SPARQL (pronounced **"sparkle"**) is a recursive acronym which stands for the **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage.
- SPARQL 1.0 W3C-Recommendation since January 15th 2008, SPARQL 1.1 W3C-Recommendation since March 21st 2013
- SPARQL 1.1 consists of a set of specifications:

Relevant to us for writing & executing queries

  - **Query language for RDF** **(how to retrieve information from RDF graphs)*** **Focus of today's session**
  - **Federated queries** (extension of query language with features for executing queries over multiple distributed RDF graphs on the Web)
  - **Graph updates** (features for how to manipulate RDF graphs e.g. inserting or deleting triples)
  - Supported **entailment regimes** ("flavours of reasoning" possible with SPARQL implementations)
  - Query results formatting JSON, XML, CSV etc. (how to represent query results in various data formats)
  - Protocol (how to communicate SPARQL queries with services/implementations that process them)
  - Service Description (How to discover SPARQL services and a vocab for describing them)
  - Graph Store HTTP Protocol (An alternative to graph updates spec - use HTTP to manipulate graphs)
  - Test Cases - A suite of tests, helpful for understanding corner cases in the specification and assessing whether a system is SPARQL 1.1 conformant

Relevant to developers of RDF graph management systems and SPARQL engines

Specifications vs. technologies / implementations

W3C specifications for SPARQL (and RDF, RDFS) are essentially **blueprints** for how to build technologies (i.e. what constraints / requirements should these technologies satisfy)

Caution: in the “Wild Wild Web”, there are many implementations claiming to comply with W3C standards. Some don’t, some do, some are more reliable than others. We will try to expose you in this course to the more established and reputable ones. If there is any doubt about W3C conformance for specific implementations, the W3C specs are the “go to” ground truth resources and there are test specs.

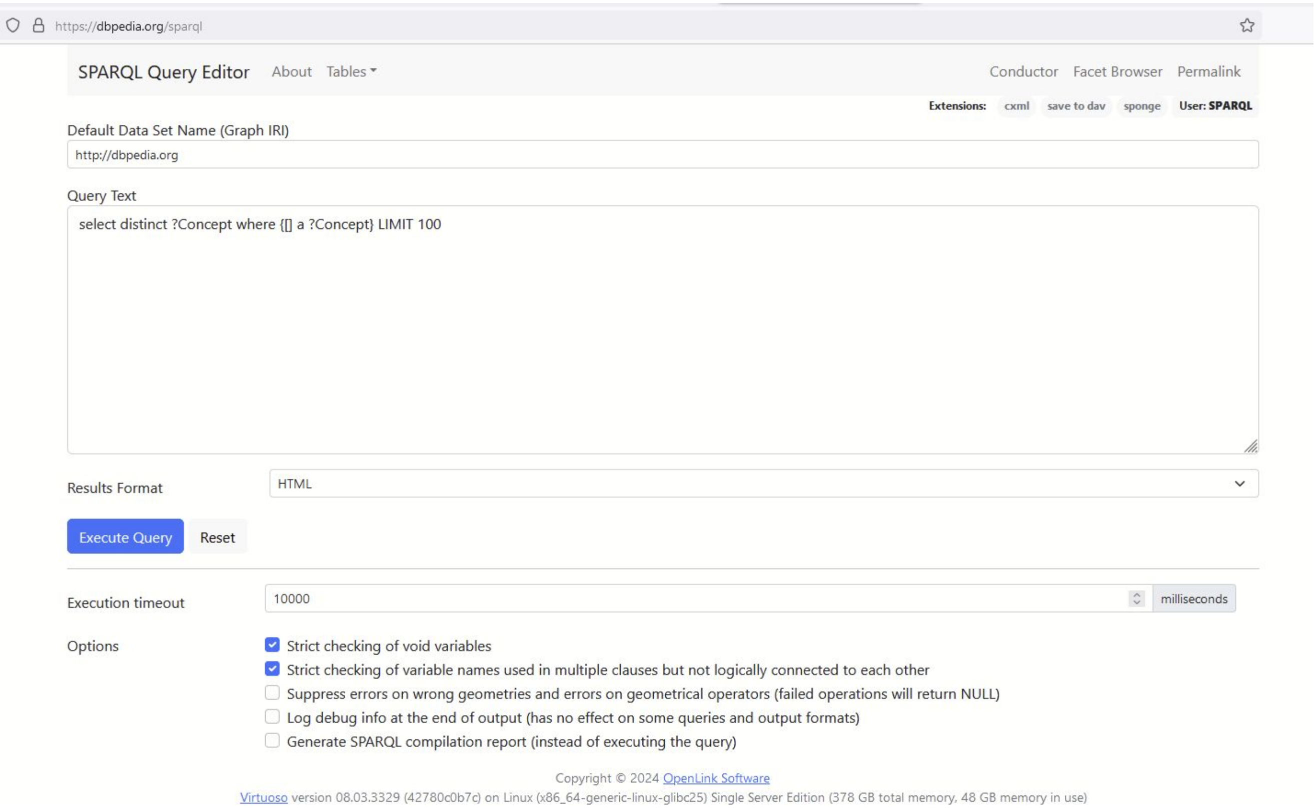
Working with large RDF graphs

- Just like the relational database (RDB) world has a host of Database Management Systems (DBMSs) e.g. MySQL, PostgreSQL, there is analogous infrastructure for RDF graphs
- Software for managing RDF graphs are often called (Semantic / RDF) graph database systems or **triplestore / quadstore implementations**. When you load / manage your RDF from these systems it is called a **triplestore / quadstore**
- These systems usually contain software for interpreting and executing SPARQL queries (called a **SPARQL engine**) and allow one to access / query triplestores by exposing a URL (called a **SPARQL endpoint**) which you can pose queries to using the **SPARQL protocols** (much like Web APIs in the RDB world).
- Alternatively, many of these systems also have web user interfaces where you can type out and execute SPARQL queries on the triplestore

Managing real-world RDF graphs: RDB vs. RDF

Relational DBs	RDF triplestores
Concept: relational database model (table relations, attributes, primary key, foreign key)	**Concept:** RDF abstract model for capturing information as triples (subject, predicate, object)
Query language: Structured Query Language (SQL)	**Query language:** SPARQL
RDBMS implementations: MySQL, PostgresQL...	**Triplestore implementations:** Virtuoso, Allegrograph, GraphDB...
Access to relational DBs: (Web) APIs provide access to DB through URL endpoints that can be queried from code or web user interfaces	**Access to RDF graphs:** SPARQL endpoint URL provides access to triplestore which can be queried from code or web user interfaces

Accessing SPARQL Endpoints



Default Data Set Name (Graph IRI)

http://dbpedia.org

Query Text

```
select distinct ?Concept where {[] a ?Concept} LIMIT 100
```

Results Format: HTML

Execution timeout: 10000 milliseconds

Options

- ☑ Strict checking of void variables
- ☑ Strict checking of variable names used in multiple clauses but not logically connected to each other
- □ Suppress errors on wrong geometries and errors on geometrical operators (failed operations will return NULL)
- □ Log debug info at the end of output (has no effect on some queries and output formats)
- □ Generate SPARQL compilation report (instead of executing the query)

Copyright © 2024 OpenLink Software

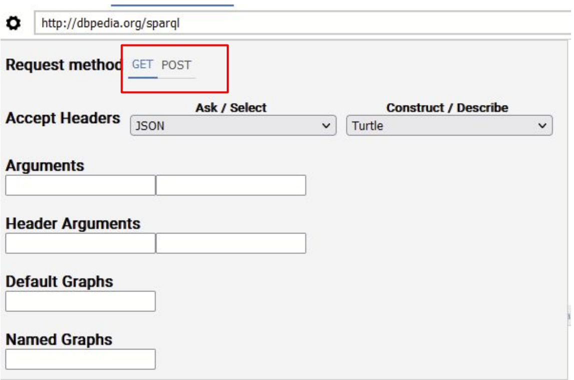
Virtuoso version 08.03.3329 (42780c0b7c) on Linux (x86_64-generic-linux-glibc25) Single Server Edition (378 GB total memory, 48 GB memory in use)

Many SPARQL endpoints provide a user interface to submit a query and view the results

SPARQL endpoint are accessible via parameterised **HTTP(S) URLs** using *HTTP GET and/or HTTP POST operation.*

YASGUI is a nice client side SPARQL query UI tool

some SPARQL endpoints accept either GET or POST, but not both



http://dbpedia.org/sparql

Request method: GET POST

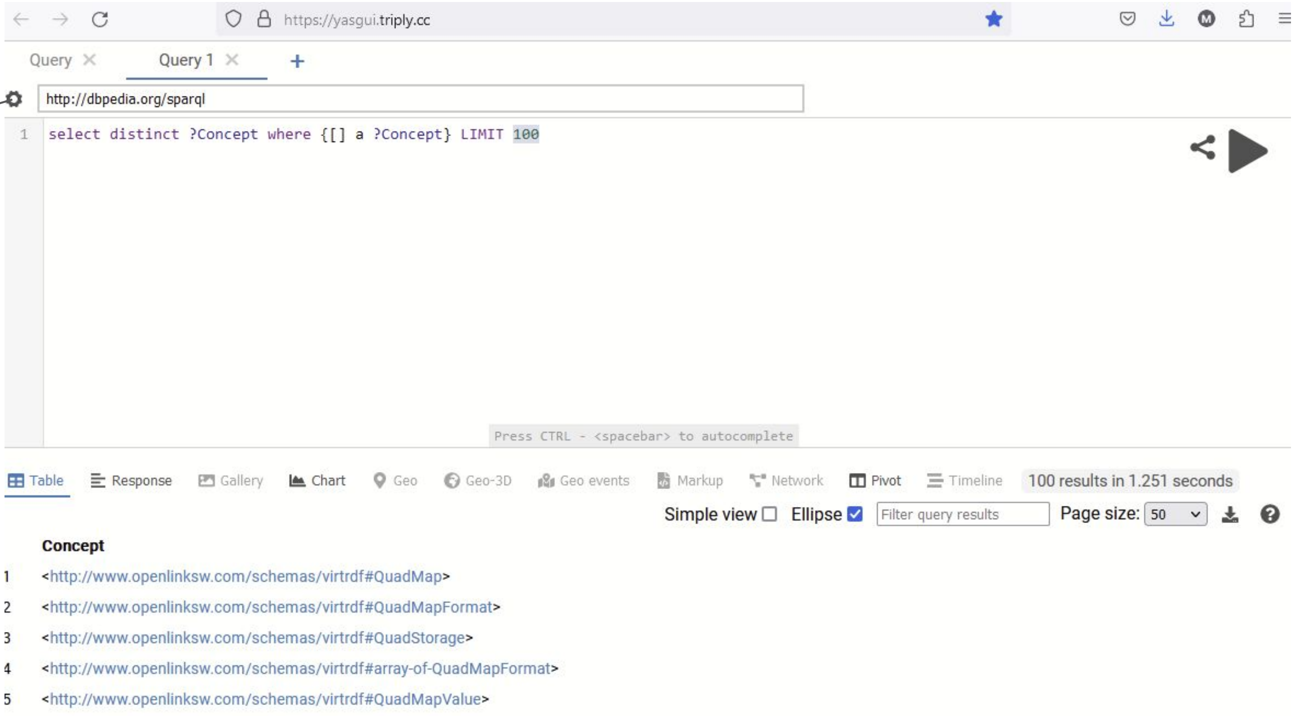
Accept Headers — Ask / Select: JSON — Construct / Describe: Turtle

Arguments

Header Arguments

Default Graphs

Named Graphs



https://yasgui.triply.cc

http://dbpedia.org/sparql

```
select distinct ?Concept where {[] a ?Concept} LIMIT 100
```

Press CTRL - <spacebar> to autocomplete

100 results in 1.251 seconds

Simple view □ Ellipse ☑ Filter query results Page size: 50

	Concept
1	<http://www.openlinksw.com/schemas/virtrdf#QuadMap>
2	<http://www.openlinksw.com/schemas/virtrdf#QuadMapFormat>
3	<http://www.openlinksw.com/schemas/virtrdf#QuadStorage>
4	<http://www.openlinksw.com/schemas/virtrdf#array-of-QuadMapFormat>
5	<http://www.openlinksw.com/schemas/virtrdf#QuadMapValue>

https://yasgui.triply.cc

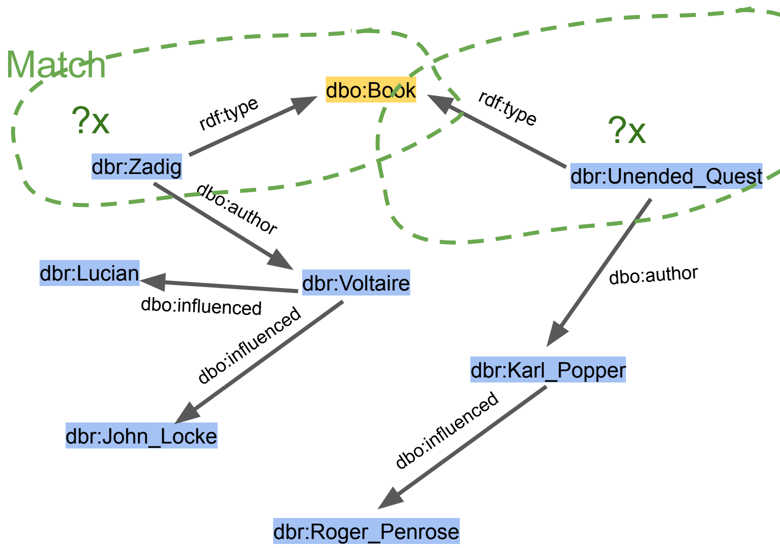
Types of SPARQL queries

Four types of operations:

- **SELECT:** Retrieve entities matching identified variables from graph pattern
- **CONSTRUCT:** create a target graph from graph pattern
- **UPDATE (INSERT/DELETE):** Add / remove triples in an RDF graph
- **ASK:** Returns a boolean answer (true/false) to specified graph pattern

SPARQL graph patterns

Main idea is to identify parts of a data graph that matches a defined graph pattern.



the WHERE { .. } clause defines a basic graph pattern

```
WHERE {
  ?x rdf:type dbo:Book .
}
```

triple pattern

constants are fixed IRIs or literals

variables are defined by a question mark "?". They can take on any literal name (e.g. ?x, ?book, ?Book).

here, the graph pattern must match any node in the graph which is the subject for in the triple pattern (?x, rdf:type, dbo:Book).

Anatomy of a SPARQL query

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

SELECT ?name ?author ?pages
WHERE {
  ?book a dbo:Book ;
    dbo:author ?author ;
    dbo:numberOfPages ?pages ;
    rdfs:label ?name .
  FILTER (?pages > 500)
  FILTER (langMATCHES(LANG(?name),"en"))
}
ORDER BY ?pages
LIMIT 10
```

Prefix declarations

Variables to display in the results

Where clause to define the basic graph pattern (BGP)
Match and filter specific triples

“Triple patterns” to match in the graph

Filter triples based on the values of some entities

Solution sequence modifiers:
Order by, group by, offset, limit clauses

Anatomy of a SPARQL query (cont...)

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

SELECT *
FROM <...>
WHERE {
  ?book a dbo:Book ;
    dbo:author ?author ;
    dbo:numberOfPages ?pages ;
    rdfs:label ?name .
  FILTER (?pages > 500)
  FILTER (langMATCHES(LANG(?name),"en"))
}
ORDER BY ?pages
LIMIT 10
```

Type of query:
SELECT,
CONSTRUCT,
INSERT ...

FROM: URI for specific subgraph you want to query

*

Return all variables in the BGP

SPARQL Triple pattern

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

SELECT *
WHERE {
  dbr:Havana_Storm a dbo:Book .
}
```

0 variables

Returns the number of occurrences of this triple in the graph

SPARQL Triple pattern

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

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

Match any triple.
Return the position-bound variables
(e.g. ?s ?p ?o)

SPARQL Triple pattern

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

SELECT *
WHERE {
  ?s ?p dbo:Book .
}
```

OR

```
SELECT *
WHERE {
   dbr:Havana_Storm ?p ?o .
}
```

OR

```
SELECT *
WHERE {
   ?s rdf:type ?o .
}
```

Match all triples
that contain the position-bound constant.
Return the remaining 2 solution bound variables

SPARQL Triple pattern

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

SELECT *
WHERE {
  ?s rdf:type dbo:Book .
}
```

OR

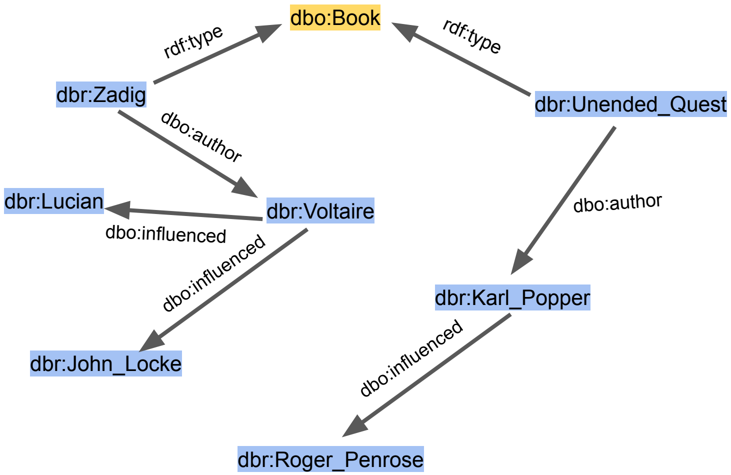
```
SELECT *
WHERE {
   dbr:Havana_Storm rdf:type ?o .
}
```

OR

```
SELECT *
WHERE {
   dbr:Havana_Storm ?p dbo:Book .
}
```

Match all triples
that contains two position-bound constants.
Return the solution bound variable.

Poll point 2: Querying an RDF graph



Which books have authors which influenced the philosopher John Locke?

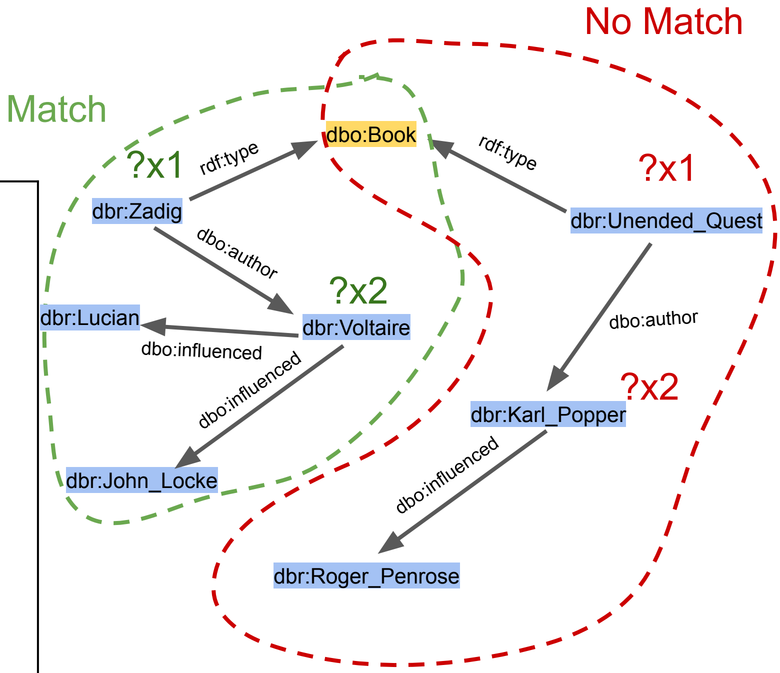
```
dbr:Zadig rdf:type dbo:Book .
dbr:Unended_Quest rdf:type dbo:Book .
dbr:Zadig dbo:author dbr:Voltaire .
dbr:Unended_Quest dbo:author dbr:Karl_Popper .
dbr:Karl_Popper dbo:influenced dbr:Roger_Penrose .
dbr:Voltaire dbo:influenced dbr:Lucian .
dbr:Voltaire dbo:influenced dbr:John_Locke
```

How did you arrive at the answer?
What “patterns” did you notice?

SPARQL graph patterns

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

SELECT ?x1
WHERE {
  ?x1 rdf:type dbo:Book .
  ?x1 dbo:author ?x2 .
  ?x2 dbo:influenced dbr:John_Locke .
}
```

(Basic) graph pattern: chain triple patterns together



Logical reading: I am looking for some entity in the graph, let’s call it “x1”. It should have an rdf:type relation to the entity “dbo:Book”. It should also have a “dbo:author” relation to some other entity, let’s call it “x2”. Finally, “x2” should have a “dbo:influenced” relation to the entity “dbr:John_Locke”. x1 and x2 can have other relations as well, but the ones mentioned above are mandatory to match the pattern.

Intuitive reading: select all books from DBpedia where the author of this book influenced John Locke.

Example SPARQL query

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT ?s

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

“All people who have male children”

Graph

```
:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:linda :hasChild :jason .

:jason schema:gender schema:Male .
```

s
:linda

:linda is a valid “binding” for ?s in this query

Example SPARQL query (*)

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT *

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

Use of * means to find bindings for EACH variable in the BGP. In simple terms: “find all possible values for ?s and ?c that satisfy this graph pattern”

Graph

```
:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:linda :hasChild :jason .

:jason schema:gender schema:Male .
```

s	c
:linda	:jason

Example SPARQL query

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT ?s

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

“All people who have male children”

Graph

```
:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:john :hasChild :sophie .

:sophie schema:gender schema:Female .

:sophie schema:sibling :george .

:george schema:gender schema:Male .

:linda :hasChild :jason .

:jason schema:gender schema:Male .
```

SPARQL reasoning

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT ?s

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

“All people who have male children ”

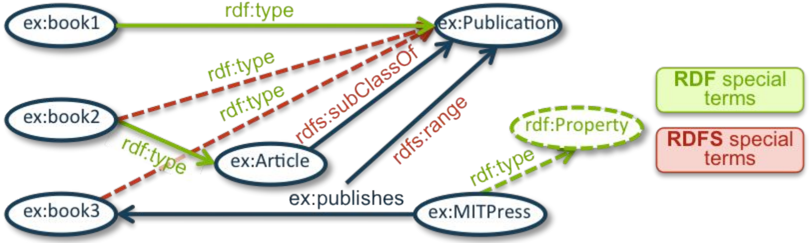
Graph

```
:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:john :hasChild :sophie .

:john :hasSon :george .

:sophie schema:gender schema:Female .

:sophie schema:sibling :george .

:george schema:gender schema:Male .

:linda :hasChild :jason .

:jason schema:gender schema:Male .

:hasSon rdfs:subPropertyOf :hasChild .
```

SPARQL 1.1 entailments

SPARQL Entailments

```
(1) ex:book1 rdf:type ex:Publication .
(2) ex:book2 rdf:type ex:Article .
(3) ex:Article rdfs:subClassOf ex:Publication .
(4) ex:publishes rdfs:range ex:Publication .
(5) ex:MITPress ex:publishes ex:book3 .
```



simple entailment only exact matches.

RDF entailment follows the RDF rules:
if: uuu aaa yyy .   (rdf1)
then: aaa rdf:type rdf:Property .

-> ex:publishes rdf:type rdf:Property

RDFS entailment follows the RDF + RDFS rules.

if: uuu rdfs:subClassOf xxx . (rdfs9)
then: vvv rdf:type uuu .
vvv rdf:type xxx .

DISTINCT & COUNT

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT ?s

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

“All people who have male children”

Graph

```
:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:linda :hasChild :jason .

:jason schema:gender schema:Male .

:linda :hasChild :luke .

:luke schema:gender schema:Male .
```

	s	**c**
1.	:linda	:jason
2.	:linda	:luke

Two possible bindings for ?s and ?c

DISTINCT & COUNT

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT ?s

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

“All people who have male children”

Graph

```
:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:linda :hasChild :jason .

:jason schema:gender schema:Male .

:linda :hasChild :luke .

:luke schema:gender schema:Male .
```

	s
1.	:linda
2.	:linda

Two possible bindings for ?s and ?c

DISTINCT & COUNT

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT DISTINCT ?s

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

"Unique people who have male children"

Graph

:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:linda :hasChild :jason .

:jason schema:gender schema:Male .

:linda :hasChild :luke .

:luke schema:gender schema:Male .

s
:linda

DISTINCT filters out the duplicates

Multiple variables with DISTINCT?

DISTINCT & COUNT

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT COUNT(?s)

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

“How many people have male children”

Graph

```
:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:linda :hasChild :jason .

:jason schema:gender schema:Male .

:linda :hasChild :luke .

:luke schema:gender schema:Male .
```

result
2

Two possible bindings for ?s and ?c

DISTINCT & COUNT

Query

```
PREFIX : <http://somenamespace.org/>

PREFIX schema: <http://schema.org/>

SELECT COUNT(DISTINCT(?s))

WHERE {

        ?s :hasChild ?c .

        ?c schema:gender schema:Male .

}
```

"How many unique people have male children"

Graph

```
:catherine :hasChild :jessica .

:jessica schema:gender schema:Female .

:linda :hasChild :jason .

:jason schema:gender schema:Male .

:linda :hasChild :luke .

:luke schema:gender schema:Male .
```

result
1

LIMIT & ORDER BY

Query

```
PREFIX : <http://somenamespace.org/>
PREFIX schema: <http://schema.org/>
SELECT ?s
WHERE {
        ?s :hasChild ?c .
        ?c schema:gender schema:Male .
}
LIMIT 1
```

“Only give 1 result for people who have male children”

Graph

```
:catherine :hasChild :jessica .
:jessica schema:gender schema:Female .
:linda :hasChild :jason .
:jason schema:gender schema:Male .
:kevin :hasChild :scott .
:scott schema:gender schema:Male .
```

s
:linda

OR

s
:kevin

LIMIT & ORDER BY

Query

```
PREFIX : <http://somenamespace.org/>
PREFIX schema: <http://schema.org/>
SELECT ?s
WHERE {
        ?s :hasChild ?c .
        ?s schema:birthDate ?dob .
        ?c schema:gender schema:Male .
}
ORDER BY DESC (?dob)
```

“People who have male children, order by date of birth youngest to oldest.”

Graph

```
:linda :hasChild :jason .
:jason schema:gender schema:Male .
:kevin :hasChild :scott .
:scott schema:gender schema:Male .
:kevin schema:birthDate “1977-05-05”^^xsd:date .
:linda schema:birthDate “1988-06-03”^^xsd:date .
```

s
:linda
:kevin

↓ Most recent to oldest

LIMIT & ORDER BY

Query

```
PREFIX : <http://somenamespace.org/>
PREFIX schema: <http://schema.org/>
SELECT ?s
WHERE {
        ?s :hasChild ?c .
        ?s schema:birthDate ?dob .
        ?c schema:gender schema:Male .
}
ORDER BY ASC(?dob)
```

“People who have male children, order by date of birth oldest to youngest”

Graph

```
:linda :hasChild :jason .
:jason schema:gender schema:Male .
:kevin :hasChild :scott .
:scott schema:gender schema:Male .
:kevin schema:birthDate “1977-05-05”^^xsd:date .
:linda schema:birthDate “1988-06-03”^^xsd:date .
```

s
:kevin
:linda

FILTER (numeric)

Query

```
PREFIX : <http://somenamespace.org/>
PREFIX schema: <http://schema.org/>
SELECT ?s
WHERE {
          ?s :hasChild ?c .
          ?c schema:gender schema:Male .
          ?c :height ?heightc .
          FILTER (?heightc > 175 && ?heightc < 190)
}
```

"People who have male children in a certain height range"

Graph

```
:linda :hasChild :jason .
:jason schema:gender schema:Male .
:kevin :hasChild :scott .
:scott schema:gender schema:Male .
:kevin :height “174”^^xsd:integer .
:linda :height “160”^^xsd:integer .
:scott :height “178”xsd:integer .
:jason :height “192”^^xsd:integer .
```

s
:kevin

FILTER (numeric & casting)

Query

```
PREFIX : <http://somenamespace.org/>
PREFIX schema: <http://schema.org/>
SELECT ?s
WHERE {
        ?s :hasChild ?c .
        ?c schema:gender schema:Male .
        ?c :height ?heightc .
        FILTER (xsd:integer(?heightc) > 175 && xsd:integer(?heightc) < 190)
}
```

“People who have male children in a certain height range”

Graph

```
:linda :hasChild :jason .
:jason schema:gender schema:Male .
:kevin :hasChild :scott .
:scott schema:gender schema:Male .
:kevin :height “174” .
:linda :height “160” .
:scott :height “178” .
:jason :height “192” .
```

s
:kevin

FILTER (string)

Query

```
PREFIX : <http://somenamespace.org/>
PREFIX schema: <http://schema.org/>
SELECT ?c
WHERE {
        ?s :hasChild ?c .
        ?c schema:gender schema:Male .
        ?c :lastname ?lastnamec .
        FILTER (?lastnamec = “Wilson”)
}
```

“male children with the last name Wilson”

Graph

```
:linda :hasChild :jason .
:jason schema:gender schema:Male .
:kevin :hasChild :scott .
:scott schema:gender schema:Male .
:linda :lastname “Wilson” .
:jason :lastname “Wilson” .
:kevin :lastname “Smith” .
:scott :lastname “Smith” .
```

s
:jason

Additional operations & functions

Math operations, datatype & casting

Math: Average number of pages in a book

```
SELECT AVG(?pages)
WHERE{
        ?book a dbo:Book .
        ?book dbo:author ?author .
        ?author dbo:numberOfPages ?pages .
}
```

Result

"322.647058823529412"^^<http://www.w3.org/2001/XMLSchema#decimal>

Will this work: write a query to find all people older than 25? **No**

```
...
<http://.../peter> <http://.../age> "36" .
...
```

No data type specified! This is just a string

datatype:

```
SELECT datatype(?age)
WHERE{
…
}
```

Results

xsd:string

Casting:

```
SELECT ?person
WHERE{
...
FILTER(xsd:integer(?age) > 25) .
}
```

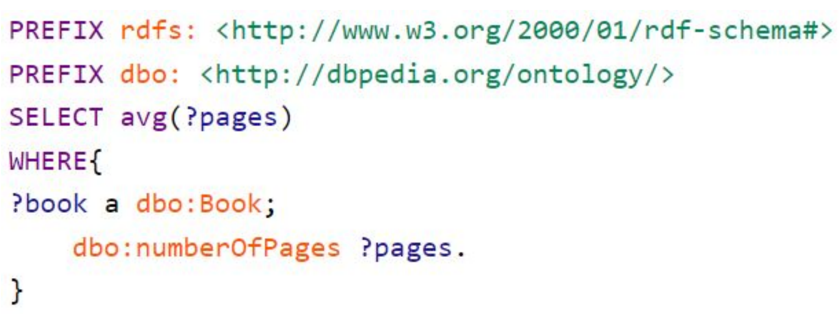
Results

peter

Additional SPARQL functions

Math operations: AVG(), SUM(),MAX(), MIN()

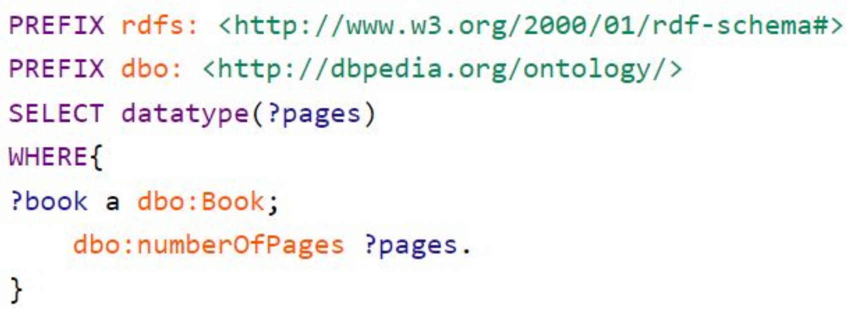
- Average number of pages



```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT avg(?pages)
WHERE{
?book a dbo:Book;
    dbo:numberOfPages ?pages.
}
```

datatype & casting

- **datatype of variable ?pages:**



```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT datatype(?pages)
WHERE{
?book a dbo:Book;
    dbo:numberOfPages ?pages.
}
```

Additional operations & functions

BIND, concat, uri

BIND:

```
SELECT ?person ?bmi
WHERE{
        ?person a schema:Person .
        ?person schema:height ?height .
        ?person schema:weight ?weight .
        BIND((?weight / (?height * ?height)) AS ?bmi) .
}
```

Result

```
Peter, 25.5
Sally, 24,3
Ken, 34.7
```

uri:

```
uri(“http://.../entity”) .
        ↓
 <“http://.../entity”>
```

concat:

```
SELECT ?fullname
WHERE{
        ?peter :firstname ?firstname .
        ?peter :lastname ?lastname .
        BIND (concat(?firstname,?lastname) AS ?fullname) .
}
```

Results

Peter Smith

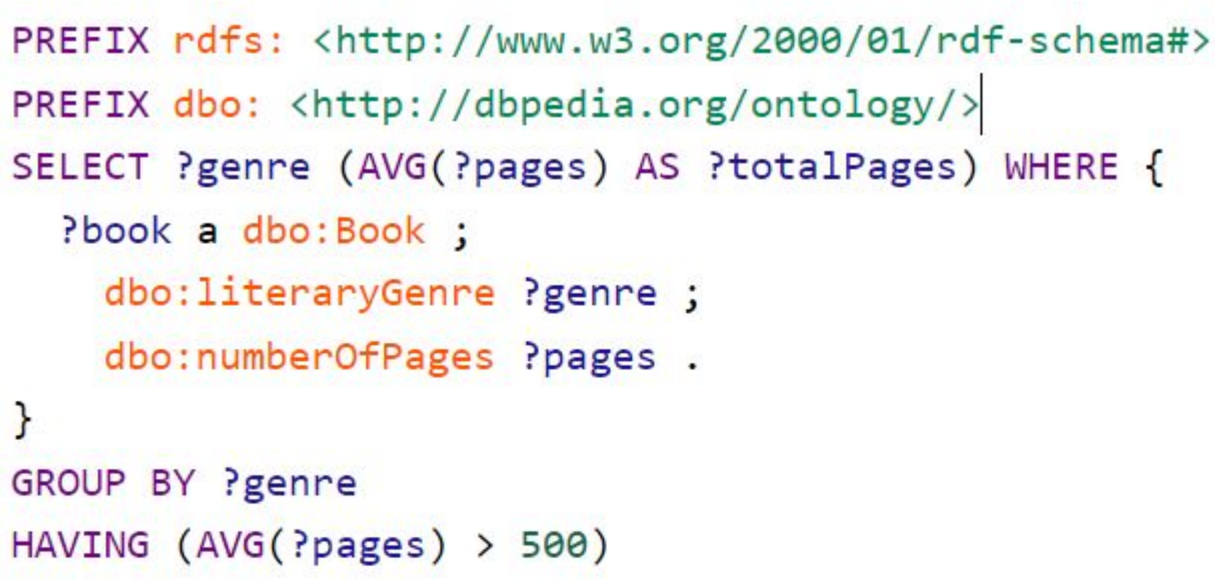
SPARQL string functions

More about other string functions (such as LCASE, STRAFTER, SUBSTR)

https://www.w3.org/TR/sparql11-query/#func-strings

Additional SPARQL functions

GROUP BY: divides results into groups and does any necessary calculations for each group
HAVING: very similar to FILTER but works on aggregated results (groups), rather than individual solutions



```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?genre (AVG(?pages) AS ?totalPages) WHERE {
  ?book a dbo:Book ;
    dbo:literaryGenre ?genre ;
    dbo:numberOfPages ?pages .
}
GROUP BY ?genre
HAVING (AVG(?pages) > 500)
```

Additional triple pattern features

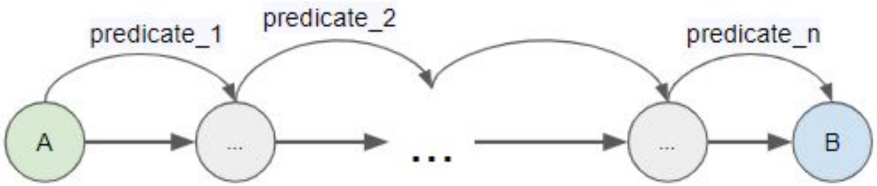
Property paths: A property path is a possible route through a graph between two graph nodes

- r | ... | s AlternativePath: Match one or both possibilities

```
{ :book1 dc:title|rdfs:label ?bookname }
```

- r / … / s SequencePath: Find an A->r->s->B chain

```
{ ?x a foaf:Person .

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



- r + … + s OneOrMorePath:

```
 {
  ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows+/foaf:name ?name .
 }
```

Additional triple pattern features

OPTIONAL: We can define optional patterns that will be retrieved when available.

Data

```
:alice  rdf:type       foaf:Person .
:alice  foaf:name      "Alice" .
:alice  foaf:mbox      <mailto:alice@example.com> .
:alice  foaf:mbox      <mailto:alice@work.example> .

:bob  rdf:type       foaf:Person .
:bob  foaf:name      "Bob" .
```

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox WHERE  {
          ?x foaf:name  ?name .
          OPTIONAL { ?x  foaf:mbox  ?mbox }
 }
```

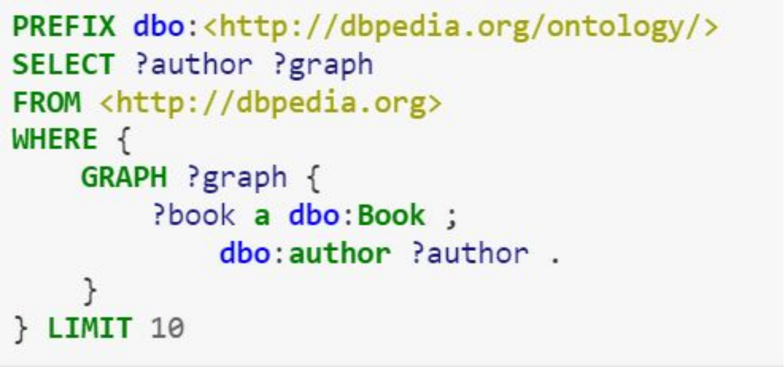
Result

name	mbox
"Alice"	<mailto:alice@example.com>
"Alice"	<mailto:alice@work.example>
"Bob"	

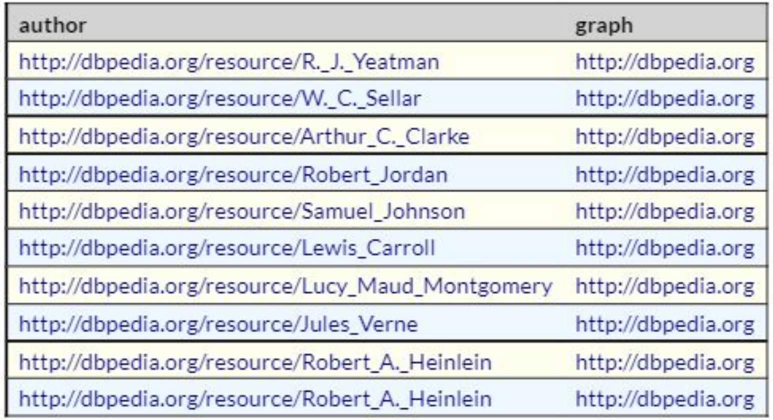
What would the result be **without** the OPTIONAL keyword?

name	mbox
"Alice"	<mailto:alice@example.com>
"Alice"	<mailto:alice@work.example>

We can query the triples FROM a specific graph:

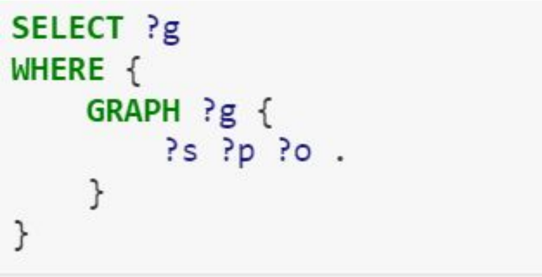


```
PREFIX dbo:<http://dbpedia.org/ontology/>
SELECT ?author ?graph
FROM <http://dbpedia.org>
WHERE {
    GRAPH ?graph {
        ?book a dbo:Book ;
            dbo:author ?author .
    }
} LIMIT 10
```

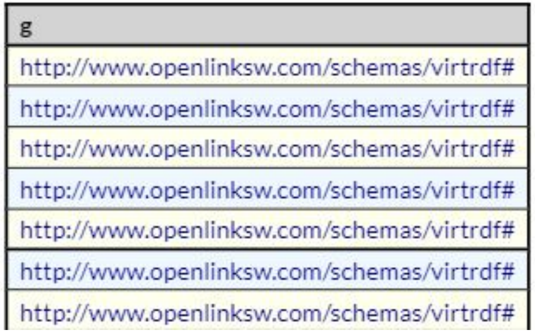


author	graph
http://dbpedia.org/resource/R._J._Yeatman	http://dbpedia.org
http://dbpedia.org/resource/W._C._Sellar	http://dbpedia.org
http://dbpedia.org/resource/Arthur_C._Clarke	http://dbpedia.org
http://dbpedia.org/resource/Robert_Jordan	http://dbpedia.org
http://dbpedia.org/resource/Samuel_Johnson	http://dbpedia.org
http://dbpedia.org/resource/Lewis_Carroll	http://dbpedia.org
http://dbpedia.org/resource/Lucy_Maud_Montgomery	http://dbpedia.org
http://dbpedia.org/resource/Jules_Verne	http://dbpedia.org
http://dbpedia.org/resource/Robert_A._Heinlein	http://dbpedia.org
http://dbpedia.org/resource/Robert_A._Heinlein	http://dbpedia.org

List all (named) subgraphs in particular endpoint



```
SELECT ?g
WHERE {
    GRAPH ?g {
        ?s ?p ?o .
    }
}
```



g
http://www.openlinksw.com/schemas/virtrdf#
http://www.openlinksw.com/schemas/virtrdf#
http://www.openlinksw.com/schemas/virtrdf#
http://www.openlinksw.com/schemas/virtrdf#
http://www.openlinksw.com/schemas/virtrdf#
http://www.openlinksw.com/schemas/virtrdf#
http://www.openlinksw.com/schemas/virtrdf#

Subqueries

```
SELECT # which variables to return
WHERE {
  { # start of the subquery
     SELECT # which variables to return
    WHERE {
      # query pattern
    }
  # query modifiers
  } # end of the subquery

# query pattern
}
# query modifiers
```

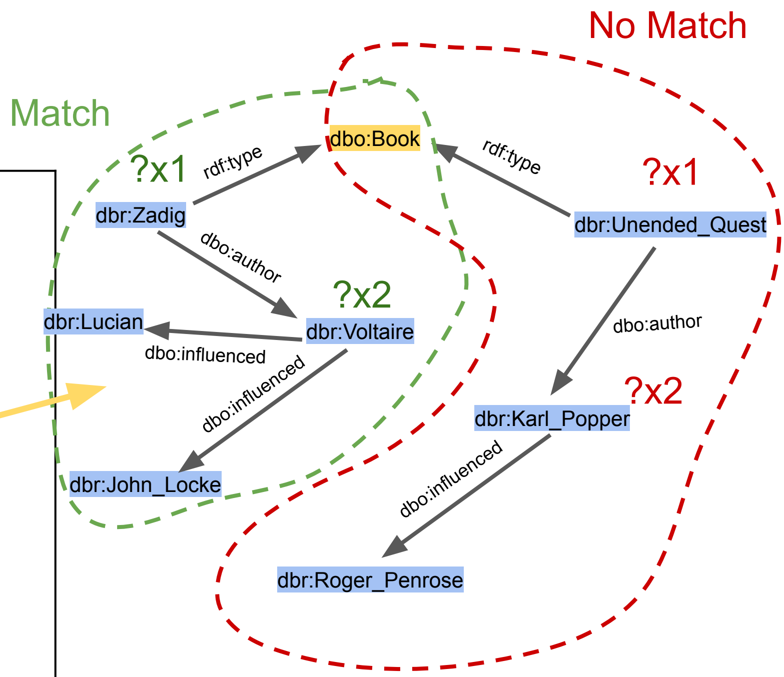
Querying the KG to select a subgraph as part of the solution and serve that subgraph to outer query

Final process for the subgraph that comes from inner query

Subqueries

```
PREFIX dbo:<http://dbpedia.org/ontology/>
PREFIX dbr:<http://dbpedia.org/resource/>
SELECT ?x1 AVG(?pages)
WHERE {
    {
      SELECT DISTINCT ?x1
      WHERE {
        ?x1 a dbo:Book .
        ?x1 dbo:author ?x2 .
        ?x2 dbo:influenced dbr:John_Locke .
      }
    }
  ?x1 dbo:numberOfPages ?pages .
  ?x1 dbo:author ?author .
}
```

Pattern matching within this subgraph



Intuitive reading: the average number of pages in books from DBpedia where the authors of the books influenced John Locke.

Example - Subqueries

*“A query **inside** a query”*

Order the **first** 10 countries that have been dissolved by date of creation.

- **Select** all countries that have been dissolved
- **Order** them by dissolution date (oldest to newest)
- **Limit** to 10
- Finally, **order** the results (10 countries) from the most recently created to the oldest created

The subquery - 10 countries from oldest to newest by their dissolution dates

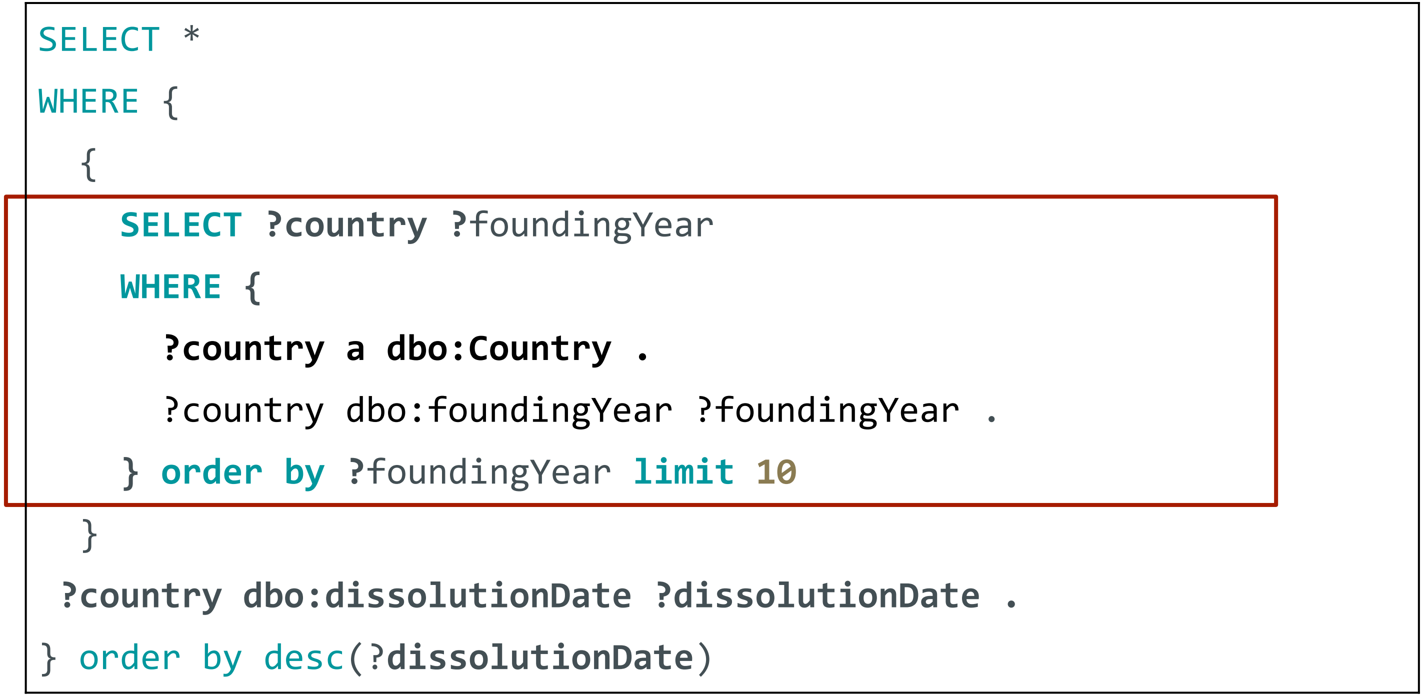
Solution

```
SELECT *
WHERE {
  {
    SELECT ?country ?dissolutionDate
    WHERE {
      ?country a dbo:Country .
      ?country dbo:dissolutionDate ?dissolutionDate .
    } order by ?dissolutionDate limit 10
  }
  ?country dbo:foundingYear ?foundingYear .
} order by desc(?foundingYear)
```

Subgraph - 10 ordered countries by their dissolution dates

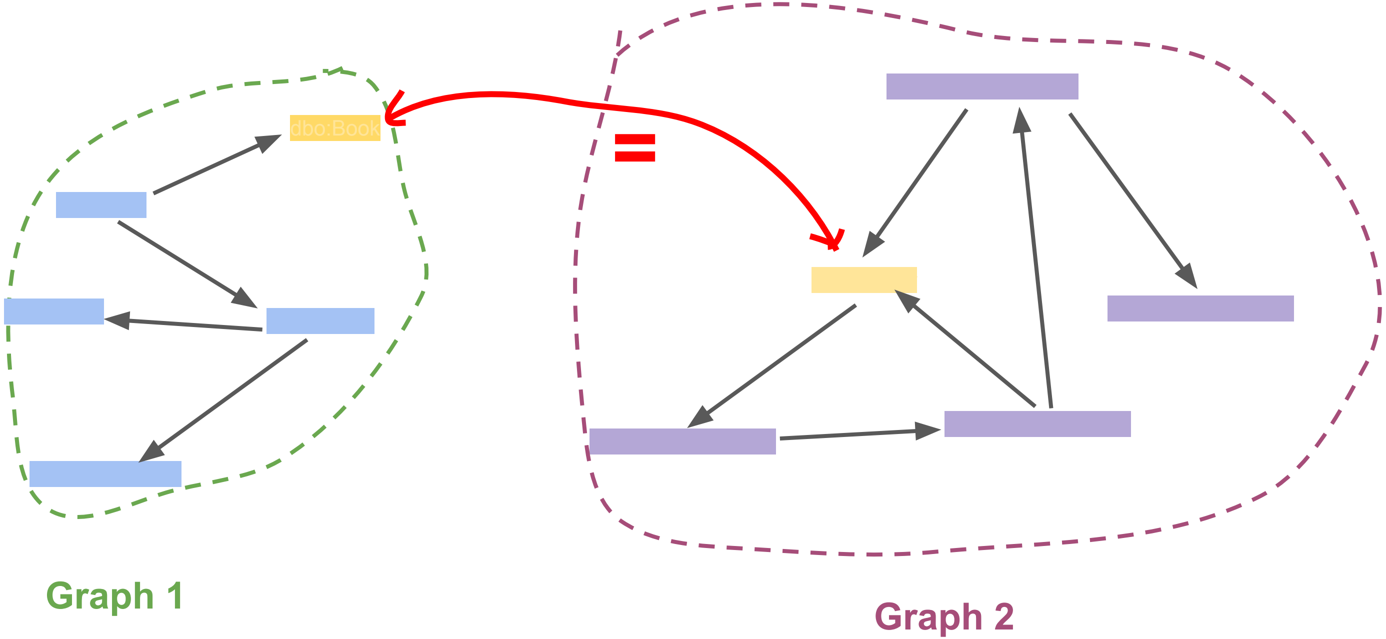
Sorted subgraph from the most recently created to the oldest created

What does this query do?



```
SELECT *
WHERE {
  {
    SELECT ?country ?foundingYear
    WHERE {
      ?country a dbo:Country .
      ?country dbo:foundingYear ?foundingYear .
    } order by ?foundingYear limit 10
  }
 ?country dbo:dissolutionDate ?dissolutionDate .
} order by desc(?dissolutionDate)
```

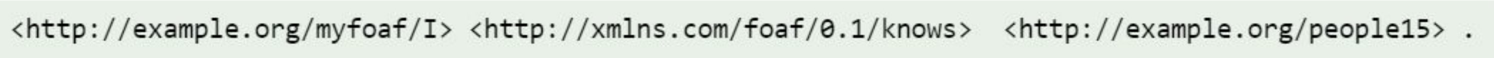
Federated queries



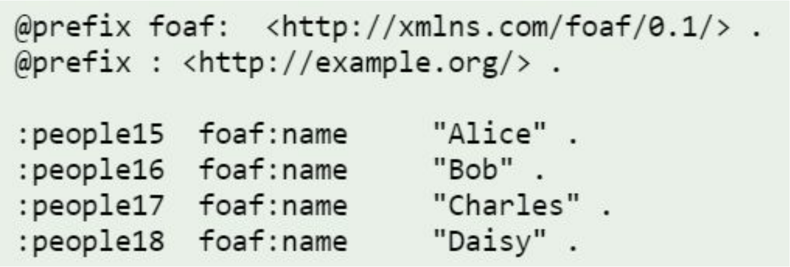
There is more info about some entity in another (external) triplestore / SPARQL endpoint. Want to query the **full extended graph** to get more information about the entity (that is not known by looking at just one of the graphs)

Federated queries

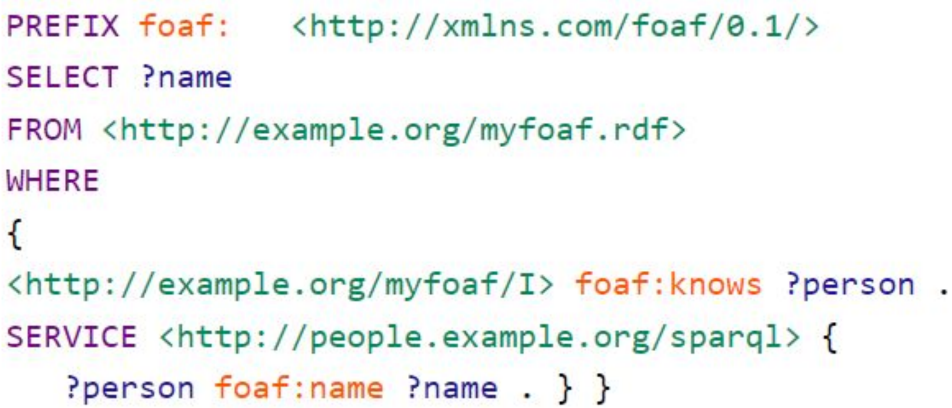
Call to a remote SPARQL endpoint to get the data



```
<http://example.org/myfoaf/I> <http://xmlns.com/foaf/0.1/knows>  <http://example.org/people15> .
```

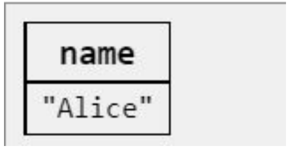


```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix : <http://example.org/> .

:people15  foaf:name     "Alice" .
:people16  foaf:name     "Bob" .
:people17  foaf:name     "Charles" .
:people18  foaf:name     "Daisy" .
```



```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM <http://example.org/myfoaf.rdf>
WHERE
{
<http://example.org/myfoaf/I> foaf:knows ?person .
SERVICE <http://people.example.org/sparql> {
    ?person foaf:name ?name . } }
```

Query Result:



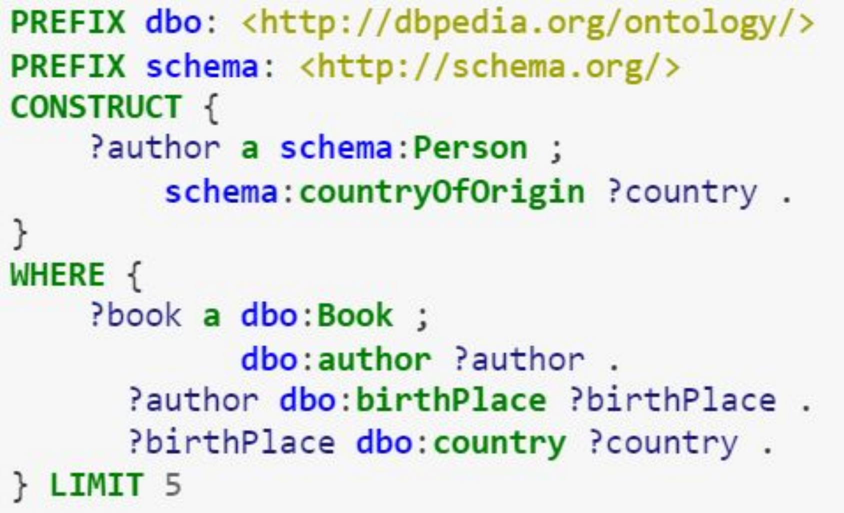
name
"Alice"

Construct, insert and delete

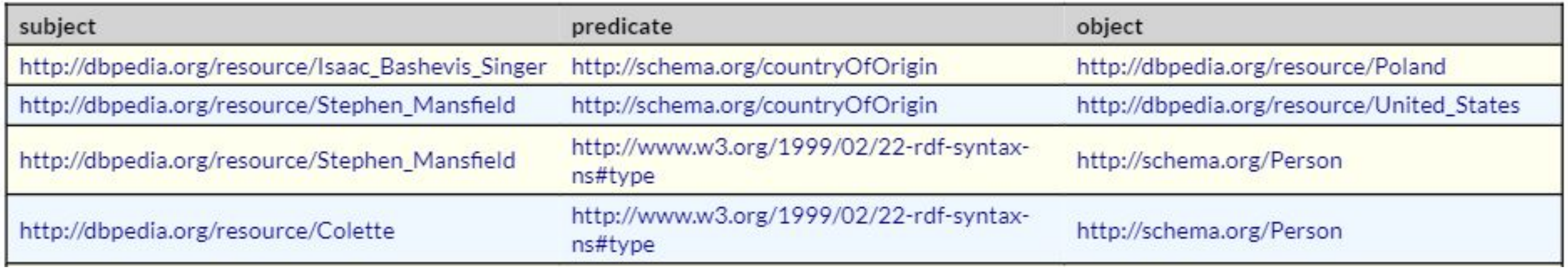
- **CONSTRUCT:** create triples and return them
- **INSERT:** creates triples and inserts the constructed triples into the (specified) graph.
- **DELETE:** similar structure to both CONSTRUCT and INSERT - deletes triples from the graph!

SPARQL query forms

Construct Creates RDF triples from matches



```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX schema: <http://schema.org/>
CONSTRUCT {
    ?author a schema:Person ;
        schema:countryOfOrigin ?country .
}
WHERE {
    ?book a dbo:Book ;
            dbo:author ?author .
      ?author dbo:birthPlace ?birthPlace .
      ?birthPlace dbo:country ?country .
} LIMIT 5
```

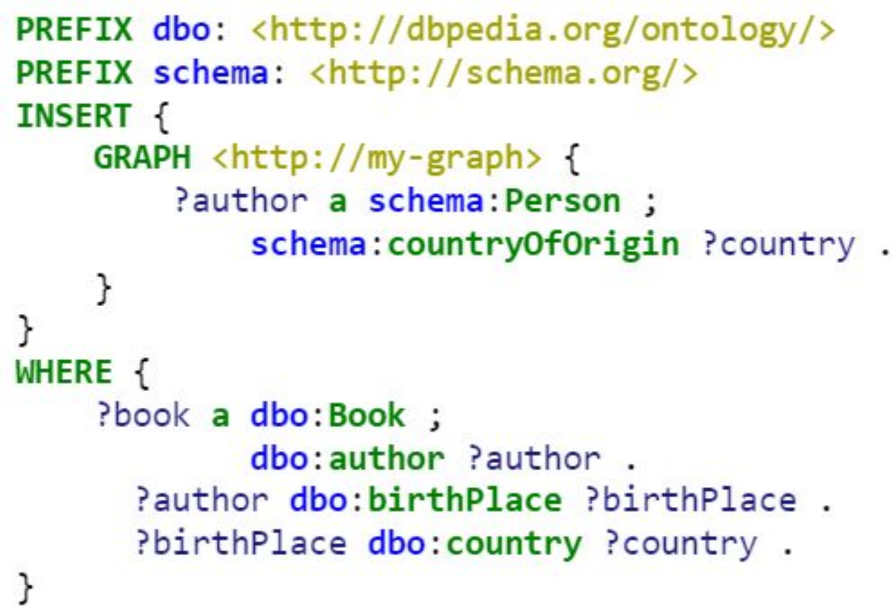


subject	predicate	object
http://dbpedia.org/resource/Isaac_Bashevis_Singer	http://schema.org/countryOfOrigin	http://dbpedia.org/resource/Poland
http://dbpedia.org/resource/Stephen_Mansfield	http://schema.org/countryOfOrigin	http://dbpedia.org/resource/United_States
http://dbpedia.org/resource/Stephen_Mansfield	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://schema.org/Person
http://dbpedia.org/resource/Colette	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://schema.org/Person

SPARQL query forms

Update: Insert 📝

Same as a construct but directly insert triples into your triplestore. You can define in which graph the triples will be inserted

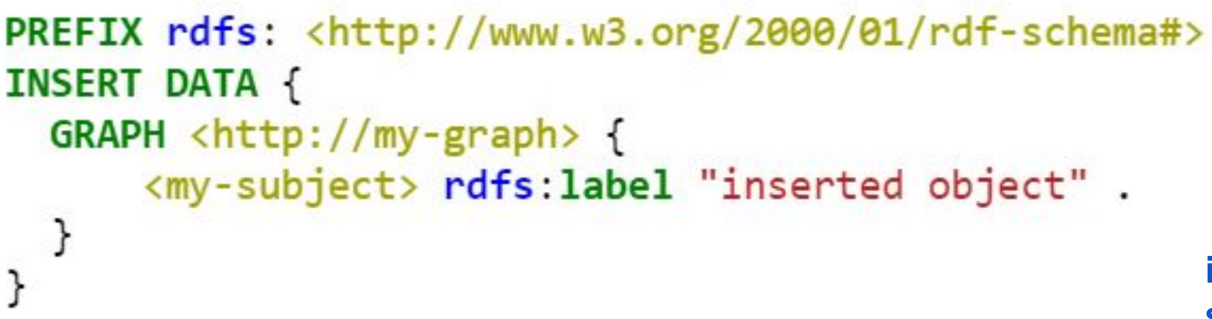


```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX schema: <http://schema.org/>
INSERT {
    GRAPH <http://my-graph> {
        ?author a schema:Person ;
            schema:countryOfOrigin ?country .
    }
}
WHERE {
    ?book a dbo:Book ;
            dbo:author ?author .
      ?author dbo:birthPlace ?birthPlace .
      ?birthPlace dbo:country ?country .
}
```

SPARQL query forms

Update: Insert DATA

Simply use SPARQL to insert data into your triplestore.
Inserts a triple pattern **with 0 variables** into the triplestore



```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
INSERT DATA {
  GRAPH <http://my-graph> {
      <my-subject> rdfs:label "inserted object" .
  }
}
```

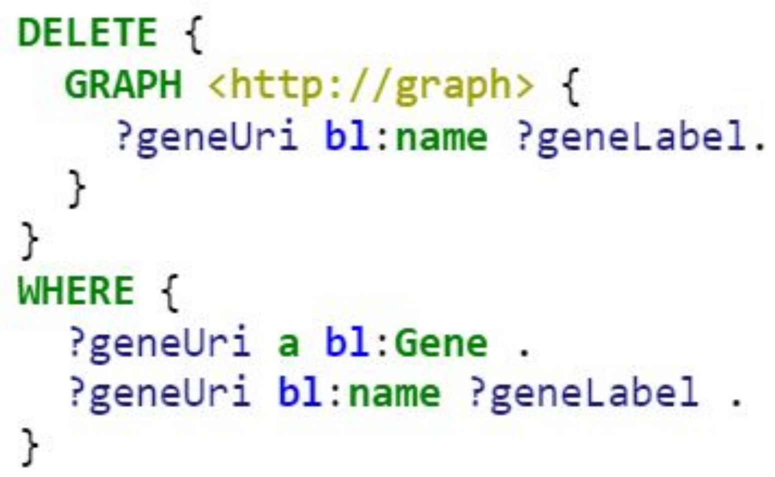
insert this exact triple in the specified graph

SPARQL query forms

Update: Delete ❌

To delete particular statements retrieved from a pattern using WHERE

Here we delete the bl:name statements for the genes we just created:



```
DELETE {
  GRAPH <http://graph> {
    ?geneUri bl:name ?geneLabel.
  }
}
WHERE {
  ?geneUri a bl:Gene .
  ?geneUri bl:name ?geneLabel .
}
```

Triple pattern to delete

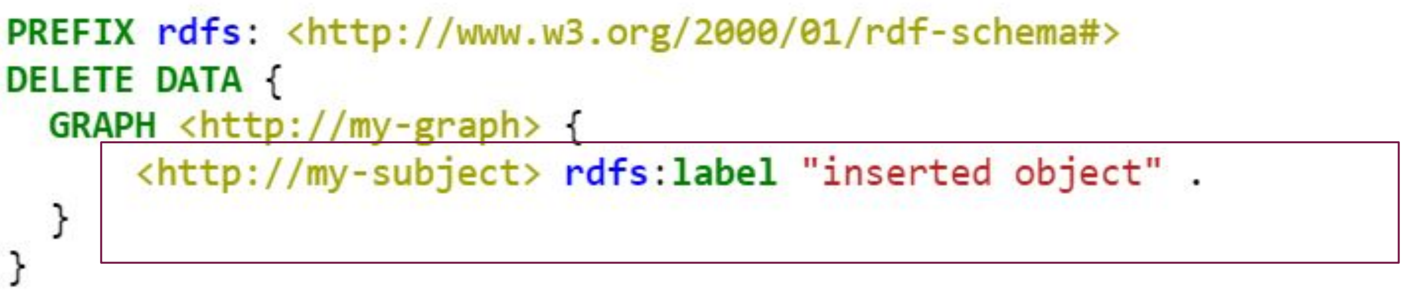
Based on the data retrieved in this where

SPARQL query forms

Update: Delete DATA

To delete particular statements. Triple patterns **with 0 variables.**

Here we delete the **rdfs:label** statements for the genes we just created



```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
DELETE DATA {
  GRAPH <http://my-graph> {
    <http://my-subject> rdfs:label "inserted object" .
  }
}
```

Delete this exact triple