The SPARQL query language for RDF

Gilles Falquet

SPARQL – based on graph patterns

Triple pattern a triple from

(RDF-Term ∪ Var) × (IRI ∪ Var) x (RDF-Term ∪ Var)

- RDF-Term : IRI or literal or blank
- Var : variable

Graph pattern

- a set of triple patterns

Same syntax as Turtle + Variables

{ ?x ex:color "red"}

{?x ex:friend ?y. ?y rdf:tye ex:Cat}

{ x:bob ex:member ?v . ?v rdf:type ex:Club . ?v ex:country ?w }

{ ?x ex:address _:adr . _:adr ex:city ex:Madrid }

SPARQL Basic Graph Pattern Query

prefix definitions
select *output variables*
from *graph*
where { *basic graph pattern* }

```
prefix foaf: <...>
select ?p
from <http://cui.unige.ch/g1>
where {
    ?p foaf:knows ?q . ?q foaf:familyName "Zep"
}
```

Definition: Basic Graph Pattern Matching

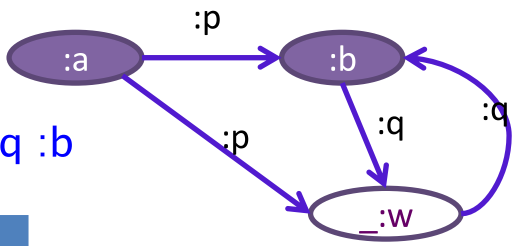
Let BGP be a basic graph pattern and let G be an RDF graph.

μ is a **solution** for BGP from G when

- there is a pattern instance mapping P such that P(BGP) is a subgraph of G
  - P maps variables and blank nodes to RDF-terms

- and μ is the restriction of P to the query variables in BGP.

Example



On the graph

:a :p :b. :a :p _:w. :b :q _:w. _:w :q :b

solutions for { ?x ?y :b }

?x	?y
:a	:p
_:w	:q

solutions for { ?x :p ?y . ?y :q ?z }

?x	?y	?z
:a	:b	_:w
:a	_:w	:b

solutions for { ?x :p _:h . _:h :q ?z }

?x	?z
:a	_:w
:a	:b

Simple Graph Patterns are not Enough

Need to express

- disjunctions (match this or that)
- optional parts in patterns (match if possible)
- negations (match this but not that)
- conditions on variable values (<, >, =, ...)
- multiple paths (path expressions) in patterns

Need to process the results

- combine the solution variables (+, -, ...)
- aggregation functions (sum, average, ...)
- ordering
- grouping

Optional parts

$$\{pattern_1 \text{ OPTIONAL } \{ pattern_2 \}\}$$

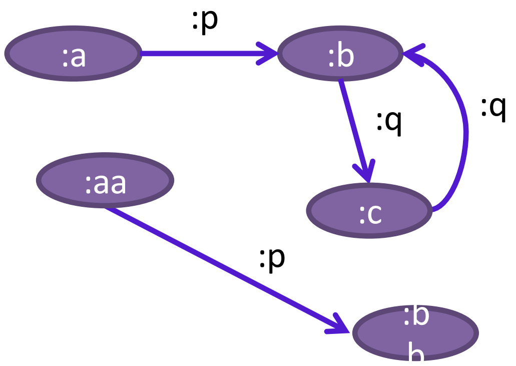
Find solutions for { $pattern_1$ $pattern_2$ } and for { $pattern_1$ }

In the solutions for $pattern_1$ only, the variables that appear in $pattern_2$ only are unbound.

Example

On the graph

:a :p :b.   :aa :p :bb.   :b :q :c.



The solutions of

{ ?x :p ?y OPTIONAL {?y :q ?z} }

are

?x	?y	?z
:a	:b	:c
:aa	:bb	UNBOUND

Union

To represent disjunctions

a solution to

pattern1 UNION *pattern2*

is a solution to *pattern1* or to *pattern2* (or both)

Example

"Find people who own a cat or a dog"

```
{?p a :Person. ?p :owns ?a. ?a a :Cat }
UNION
{?p a :Person. ?p :owns ?a. ?a a :Dog }
```

can be simplified by using group graph patterns

```
{?p a :Person. ?p :owns ?a. {{?a a :Cat} UNION {?a a :Dog}}}
```

Filtering with a boolean exprression

{*pattern* FILTER(*expression*) }

retain only the solutions to *pattern* for which *expression* evaluates to true

Example

```
{ ?x a :Car. ?x :price ?p. ?x :category ?c
        FILTER(?p < 10000 && ?c != :sport) }
```

Testing For the Absence of a Pattern

Data:

```
:alice rdf:type foaf:Person . :alice foaf:name "Alice" .
:bob rdf:type foaf:Person .
```

Query:

```
SELECT ?person
WHERE { ?person rdf:type foaf:Person .
          FILTER NOT EXISTS { ?person foaf:name ?name } }
```

Query Result:

:bob

Testing For the Presence of a Pattern

Query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person
WHERE { ?person rdf:type foaf:Person .
   FILTER EXISTS { ?person foaf:name ?name } }
```

Query Result:

```
<http://example/alice>
```

Removing Possible Solutions

MINUS evaluates both its arguments, then calculates solutions in the left-hand side that are not compatible with the solutions on the right-hand side.

```
:alice foaf:givenName "Alice" ; foaf:familyName "Smith" .
:bob foaf:givenName "Bob" ; foaf:familyName "Jones" .
:carol foaf:givenName "Carol" ; foaf:familyName "Smith" .
```

```
SELECT DISTINCT ?s
WHERE { ?s ?p ?o . MINUS { ?s foaf:givenName "Bob" . } }
```

Results:

:carol
:alice

Relationship and differences between NOT EXISTS and MINUS

NOT EXISTS and **MINUS** represent two ways of thinking about negation

- one based on testing whether a pattern exists in the data, given the bindings already determined by the query pattern,
- one based on removing matches based on the evaluation of two patterns. In some cases they can produce different answers.

```
@prefix : <http://example/> .
:a :b :c .
```

SELECT * { ?s ?p ?o FILTER NOT EXISTS { ?x ?y ?z } }

No solutions because { ?x ?y ?z } matches given any ?s ?p ?o

SELECT * { ?s ?p ?o MINUS { ?x ?y ?z } }

There is no shared variable between the first part (?s ?p ?o) and the second (?x ?y ?z) so no bindings are eliminated.

Results:

:a :b :c

Property path

iri	
^elt	inverse path
elt / elt	sequence
elt | elt	alternative
elt*	repetition (0...n)
elt+	repetition (1...n)
elt?	option
!*iri* or !(iri_1| ...|iri_n)	negation
!^*iri* or !(^iri_1| ...|^iri_n)	negation of the inverse
!(iri_1| ...|iri_j|^iri_{j+1}| ...|^iri_n)	

Using property path to access lists

Recall that a list structure, written as

```
:france :flagColors (:blue :white :red) .
```

is an abbreviation for:

```
:france :flagColors [
    rdf:type rdf:List ;
    rdf:first :blue ;
    rdf:rest [
        rdf:type rdf:List ;
        rdf:first :white ;
        rdf:rest [
            rdf:type rdf:List ;
            rdf:first :red ;
            rdf:rest rdf:nil ]]]
```

Some queries over such structures can only be solved by using property path expressions.[1]

Find all the colors in the french flag

```
select ?c
where {:france :flagColors/rdf:rest*/rdf:first ?c }
```

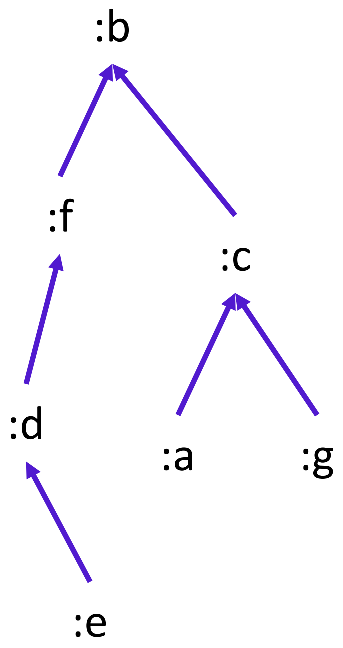
Find the last color of the french flag

```
select ?c
where {:france :flagColors/rest* ?last.
           ?last rdf:rest rdf:nil. ?last rdf:first ?c }
```

1. Or with entailment regimes that take into account transitive properties

Accessing Trees

Example: a part is decomposed into subparts, sub-subparts, etc. linked through a :partOf property.



Display all the parts that belong to :b

```
select ?p where {?p :partOf* :b. }
```

If part :a belongs to part :b, display its part number

```
select ?pn where {:a :partOf* :b. :a :partNo ?pn }
```

What are the subparts of :b that have more than 10 subparts (at any level)

```
select ?p where {?p :partOf* :b.
filter count(select ?q where {?q partOf* ?p}) > 10 }
```

Entailment Regimes

Several entailment regimes for RDF, to infer new triples

basic RDF entailment

- e.g. (x p y) → (p rdf:type rdf:Property)

RDFS entailment

- e.g. (x p y), (p rdfs:range z) → (y rdf:type z)

OWL entailment

OWL direct semantics

...

SPARQL answers should take into account inferred triples

```
(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 .
```

SELECT ?prop WHERE { ?prop rdf:type rdf:Property }

Simple entailment → ø

RDF entailment →

using RDF the entailment rule

$(x\ p\ y) \rightarrow (p$ rdf:type rdf:Property)

?prop
rdf:type
rdfs:subClassOf
rdfs:range
ex:publishes

```
(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 .
```

SELECT ?pub WHERE { ?pub rdf:type ex:Publication }

with RDF entailment →

?pub
ex:book1

with RDFS entailment →

using the rules

(x a C), (C rdfs:subClassOf D) → (x a D)

(x p y), (y rdfs:range C) → (y a D)

?pub
ex:book1
ex:book2
ex:book3

Blank nodes in graphs and results

```
ex:MITPress
  ex:published ex:bk1 ;
  ex:published _:2 .
```

- Blank nodes are local
- They have no URI
- They cannot be "exported" to the answer
- The answer mapping must "invent" blank nodes

```
select ?pub where {ex:MITPress ex:published ?pub}
```

Infinitely many possible answers ?

?pub
ex:every
_:cx323322

?pub
ex:every
_:abc

?pub
ex:every
_:u123

Scoping Graph – to avoid infinite answers

- Since SPARQL treats blank node identifiers in a results format document as scoped to the document, they cannot be understood as identifying nodes in the active graph of the dataset.

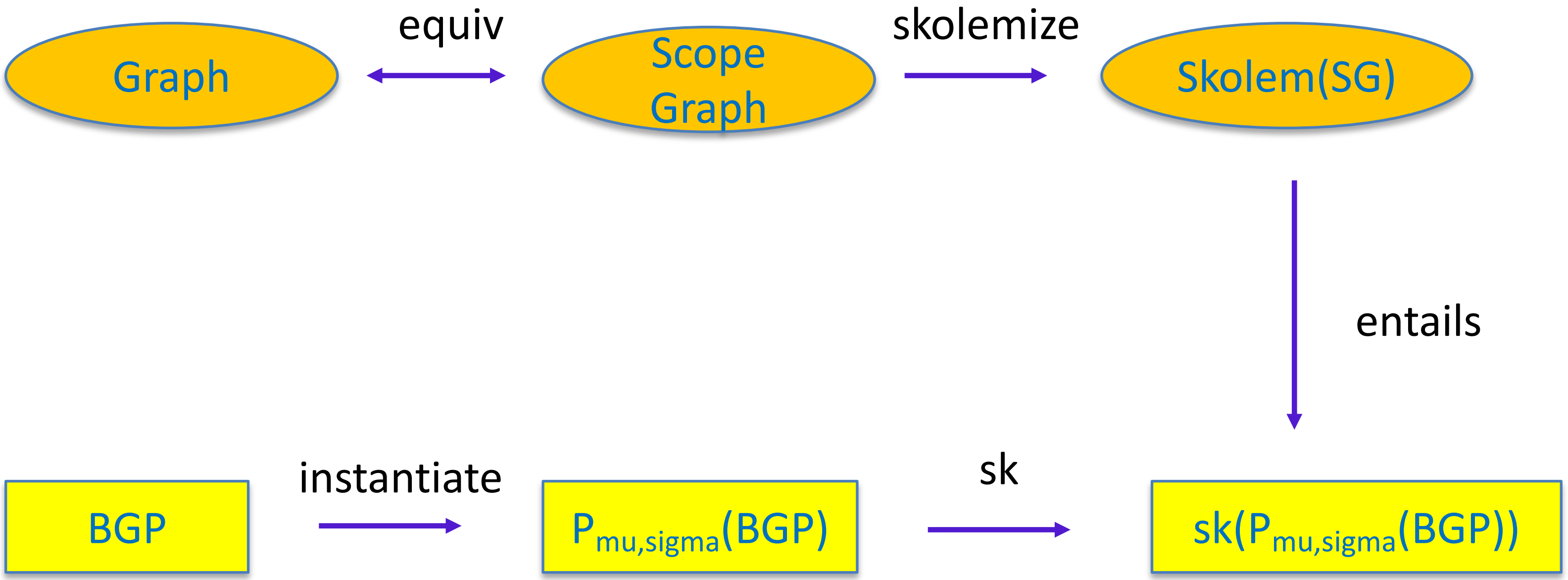
- If DS is the dataset of a query, pattern solutions are therefore understood to be not from the active graph of DS itself, but from an RDF graph, called the *scoping graph*, which is graph-equivalent to the active graph of DS but shares no blank nodes with DS or with BGP.

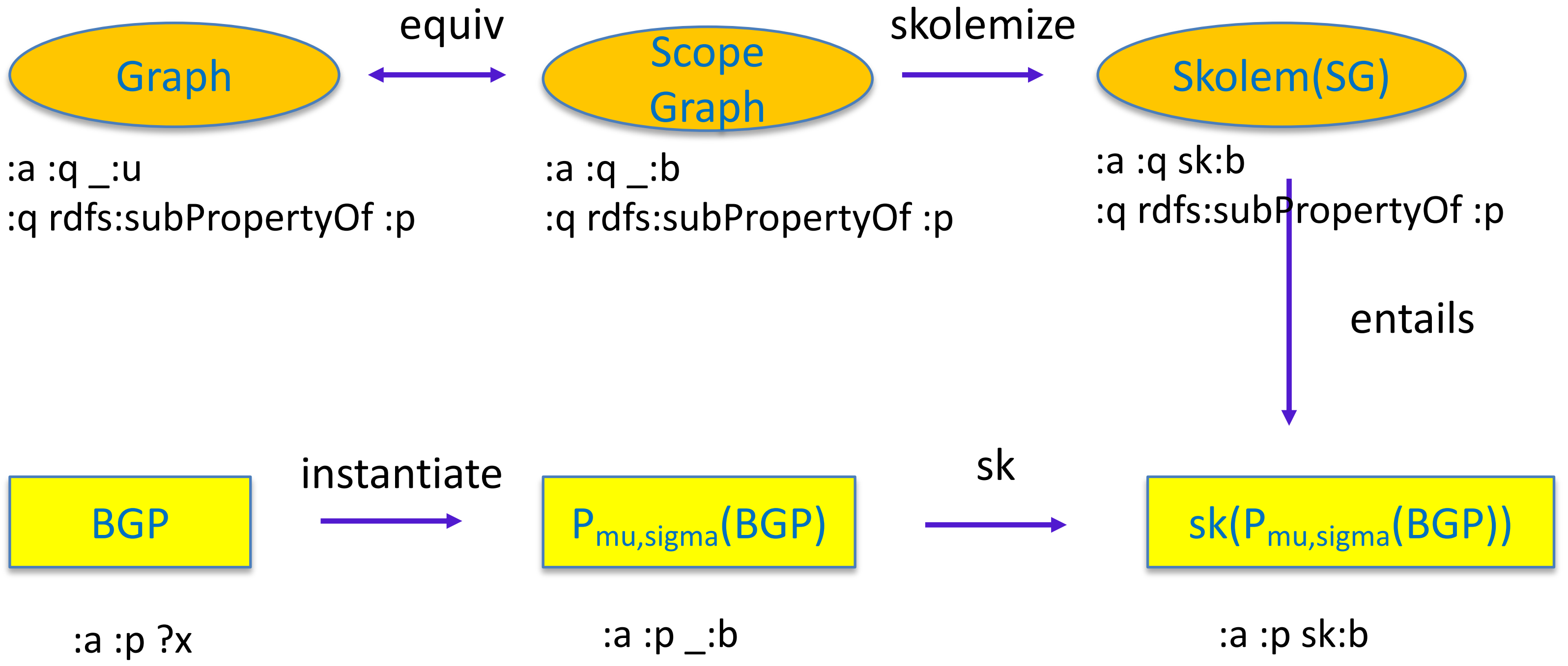
- The same scoping graph is used for all solutions to a single query.

Technical definition of a solution for a BGP

- A solution mapping μ is a *possible solution for BGP from G under RDF entailment* if
  - dom(μ) = Variables(BGP) and
  - there is an RDF instance mapping σ from B(BGP) to RDF-T such that dom(σ)=Blanks(BGP) and
  - the pattern instance mapping P=(μ, σ) is such that P(BGP) are well-formed RDF triples that are RDF(S) entailed by SG.
- A possible solution μ is a *solution for BGP from SG under RDF entailment* if:
  - (C1) The RDF triples sk(P(BGP)) are ground and RDF(S) entailed by skolem(SG).
  - (C2) For each variable x in Variables(BGP), μ(x) occurs in SG or in rdfV-Minus.

- The multiplicity of μ in the multiset of solutions is the maximal number of distinct RDF instance mappings σ that yield a pattern instance mapping P = (μ, σ) for which μ is a solution.


Graph
equiv
Scope Graph
skolemize
Skolem(SG)
entails
BGP
instantiate
P mu,sigma(BGP)
sk
sk(P mu,sigma(BGP))


Graph
equiv
Scope Graph
skolemize
Skolem(SG)
:a :q _:u
:q rdfs:subPropertyOf :p
:a :q _:b
:q rdfs:subPropertyOf :p
:a :q sk:b
:q rdfs:subPropertyOf :p
entails
BGP
instantiate
$P_{mu,sigma}(BGP)$
sk
$sk(P_{mu,sigma}(BGP))$
:a :p ?x
:a :p _:b
:a :p sk:b

Practically

- The answers are computed as if blank nodes where replaced by URI nodes

  _:b1 ---------> *new*:b1

  where *new* is a URI that does not appear in the graph