

## 0.1 Querying Semantic Web – SPARQL

### History of RDF Query Languages

**relational-based** – SPARQL, RQL, TRIPLE, Xcerpt, SeRQL

**reactive-rule language Algea** (see <http://www.w3.org/2001/Annotea>)

- *actions* (ask, assert, fwrule), *answers* (bindings for vars, **proof RDF triples**)

**path-based language Versa** (see <http://4suite.org>)

- “XPath for RDF”
- forward/backward traversal, filtering, but no support for restructuring/constructing queries.

..., there are plenty of them, but today **SPARQL wins**.

### SPARQL vs. SQL

First, let’s shortly compare a query in SQL and SPARQL.

’Get projects having male administrators starting on the letter N’

```
SELECT e.surname AS es,
       p.name AS pn
FROM employee e, project p
WHERE e.gender = 'male'
      AND p.administratorId = e.id
      AND e.surname LIKE 'N\%';
```

```
PREFIX : <http://example.org/>
SELECT ?sn, (?projname AS ?pn)
WHERE {
  ?e a :Employee .
  ?e :surname ?sn .
  ?e :gender 'male' .
  ?p a :Project .
  ?p :name ?pn .
  ?p :administrator ? e .
  FILTER (strstarts(?sn,'N'))
}
```

### SPARQL Factsheet

- SPARQL 1.1 was standardized as a set of 12 W3C Recommendations on 21 March 2013, covering
  - a query language (SPARQL 1.1 Query Language) [?]
  - a “data definition language” (SPARQL 1.1. Update language)
  - definition of SPARQL services (protocol over HTTP, graph management HTTP protocol), semantic description,

- an extension for executing distributed queries over more SPARQL endpoints [?]
- JSON, CSV, TSV, XML query result formats [?]
- definition of entailment regimes for RDF extensions (e.g. OWL, see the respective lecture) [?].

### 0.1.1 SPARQL Query Language

#### Query Types

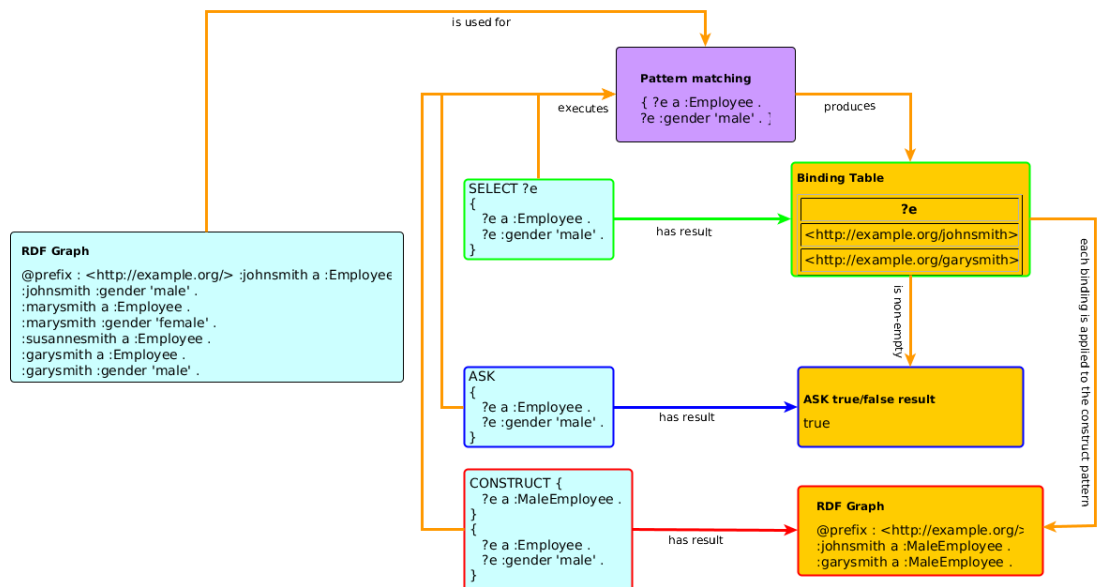
**SELECT** - returns a binding table (similarly to SQL)

**ASK** - returns a true/false indicating existence of the given pattern in the RDF graph

**CONSTRUCT** - returns an RDF graph constructed from the binding table

**DESCRIBE** - returns an RDF graph describing the given resource (semantics not fixed)

#### Query Evaluation



#### Basic Definitions

**RDF Term** is an element of the set of RDF terms  $T = T_I \cup T_B \cup T_L$ , being a union of set of all IRIs, blank nodes and literals respectively.

**graph store** is a mutable container providing an RDF dataset at each time,

**solution** is a mapping  $\mu : V \rightarrow T$  assigning an RDF term to each variable from the query,

**result set** is a list  $R = (\mu_1, \dots, \mu_n)$  of solutions,

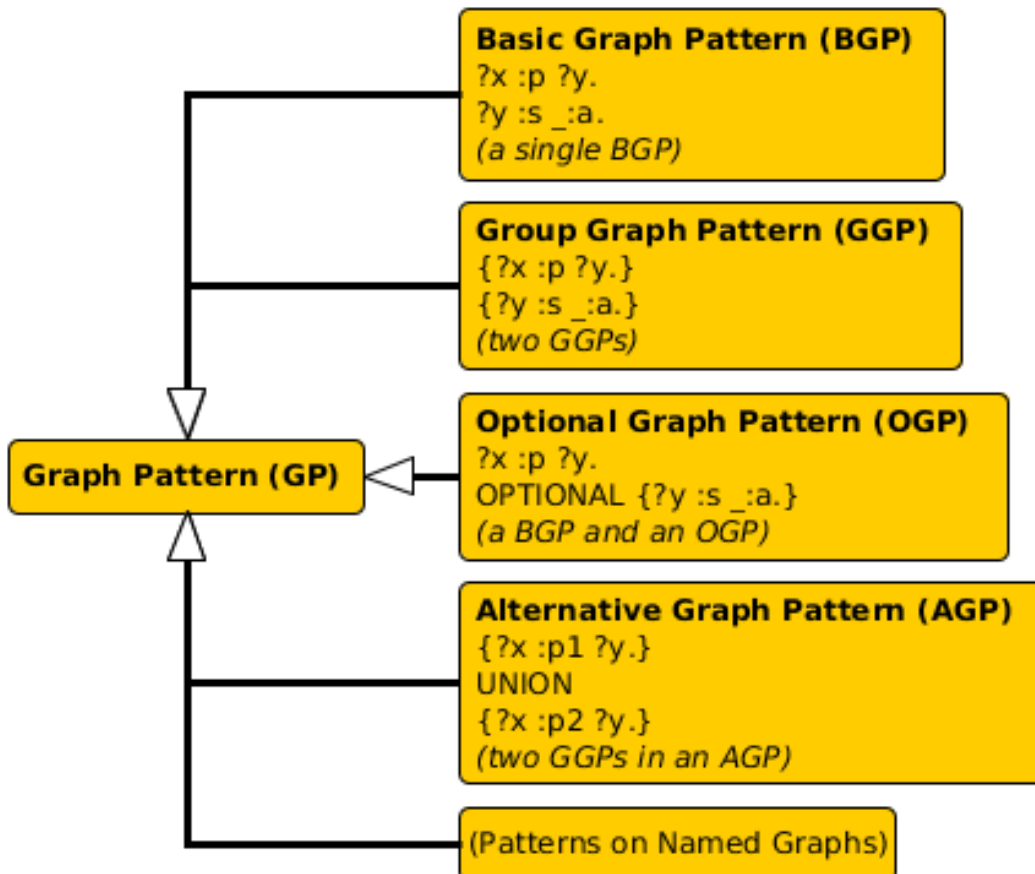
**triple pattern (TP)** is a member of  $(T \cup V) \times (T \cup V) \times (T \cup V)$ ,

**basic graph pattern (BGP)** is a set  $BGP = \{TP_1, \dots, TP_n\}$  of triple patterns.

### Graph Patterns – Overview

Graph patterns cover all basic algebraic operations:

- conjunction (sequence of graph patterns),
- disjunction (**UNION** pattern),
- negation (**FILTER NOT EXISTS**, **MINUS**)
- conditional conjunction (**OPTIONAL**)



## Basic Graph Patterns

*Listing 1 : Repository content*

```
@prefix : <http://example.org/>
@prefix r: <http://dbpedia.org/resource/>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
:inventors {
  r:Thomas_Edison :invented :bulb .
  r:J_Cimrman :invented :bulb .
  :bulb rdfs:label "Bulb"@en , "Zarovka"@cs .
  :wheel rdfs:label "Wheel"@en .
  _:x :invented :wheel .
  _:y :invented :SteamEngine .
  _:z :invented :Gunpowder .
  :Gunpowder rdfs:label "Strelny_prach"@cs .
}
```

*Listing 2 : Query with a BGP*

```
PREFIX : <http://example.org/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?s ?l
WHERE {
  ?s :invented ?i.
  ?i rdfs:label ?l.}

```

Table 0.1: Result set

| s               | l                  |
|-----------------|--------------------|
| r:Thomas_Edison | "Bulb"@en          |
| r:J_Cimrman     | "Bulb"@en          |
| r:Thomas_Edison | "Zarovka"@cs       |
| r:J_Cimrman     | "Zarovka"@cs       |
| _:a             | "Wheel"@en         |
| _:b             | "Strelny prach"@cs |

## Filtering results

### Description

**syntax** BGP1 **FILTER**(boolean condition) BGP1

**description** **FILTER** clause filters BGP results; it can be anywhere in a BGP (does not break it)

*Listing 3 : Query with a BGP*

```
PREFIX : <http://example.org/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?s ?l
WHERE {
  ?s :invented ?i.
  ?i rdfs:label ?l
  FILTER(regex(?l, "^\.ul\.*)"
    && contains(str(?s), "Cimr"))
}
```

- string functions – e.g. `strlen`, `contains`, `substr`, `concat`, `regex`, `replace`
- RDF term functions – e.g. `isIRI`, `IRI`, `isBlank`, `BNODE`, `isLiteral`, `str`, `lang`, `datatype`
- ..., see SPARQL 1.1 spec.

## Optional data

### Description

**syntax** GP1 **OPTIONAL** { GP2 }

**description** results of GP1 are optionally augmented with results of GP2, if any. Optionals are left-associative.

Listing 4 : Two optionals

```
PREFIX : <http://example.org/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?s ?i ?l
WHERE {
  ?s :invented ?i.
  OPTIONAL {
    ?i rdfs:label ?l FILTER (lang(?l)="en") .
  } OPTIONAL {
    ?i rdfs:label ?l FILTER (lang(?l)="cs")
  }
}
```

Table 0.2: Result set

| s               | l                  |
|-----------------|--------------------|
| r:Thomas.Edison | "Bulb"@en          |
| r:J.Cimrman     | "Bulb"@en          |
| ..a             | "Wheel"@en         |
| ..b             |                    |
| ..c             | "Strelny prach"@cs |

## Negation

negation as failure – i.e. what cannot be inferred is considered false.

two constructs – **MINUS** vs. **FILTER NOT EXISTS**

Listing 5 : MINUS

```
PREFIX : <http://example.org/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?s1 ?i
{ ?s1 :invented ?i.
  MINUS {
    ?s2 :invented ?i .
    FILTER(?s1 != ?s2) . }}
}
```

Variable ?s1 is not bound in the **MINUS** pattern. Returns all inventors.

*Listing 6 : FILTER NOT EXISTS*

```
PREFIX : <http://example.org/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?s1 ?i
{
  ?s1 :invented ?i.
  FILTER NOT EXISTS {
    ?s2 :invented ?i .
    FILTER(?s1 != ?s2). } }
```

Returns all inventions that were invented just by one inventor.

## Property Paths

### Description

Property paths allow to express simple regular expressions on properties, as follows

| syntax                 | matches ( $e_{(i)}$ means path element, $p_{(i)}$ means $iri$ or $\hat{iri}$ ) |
|------------------------|--|
| $iri$                  | an IRI (path of length 1)  |
| $\hat{e}$              | an inverse path ( $o \rightarrow s$ )  |
| $e_1 / e_2$            | a sequence path of $e_1$ followed by $e_2$                                     |
| $e_1   e_2$            | an alternative path of $e_1$ or $e_2$  |
| $e^*$                  | a sequence path of zero or more matches of $e$                                 |
| $e^+$                  | a sequence path of one or more matches of $e$                                  |
| $e^?$                  | a sequence path of zero or one more matches of $e$                             |
| $!(p_1   \dots   p_n)$ | any IRI not matching any of $p_i$  |
| $(e)$                  | group path (brackets for precedence)   |

## Property Paths – Examples

*Listing 7 : Get the name of a resource.*

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT *
{
  ?s rdfs:label/dc:title ?name.
}
```

*Listing 8 : Get elements of an RDF collection.*

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT *
{
  ?s (rdf:rest*)/rdf:first ?listItem.
}
```

## Aggregations

### Description

Similarly to SQL, SPARQL allows using aggregation functions for numeric/string data:

**COUNT(?var)**, or **COUNT(DISTINCT ?var)** – counts number of (distinct) occurrences of ?var in the resultset,

**MIN(?v)**, **MAX(?v)**, **SUM(?v)**, **AVG(?v)** – analogous to their SQL counterparts,

**GROUP\_CONCAT(?var; separator = <SEP> AS ?group)** – concatenates all elements in the group with the given separator character,

**SAMPLE** – takes an arbitrary representative from the group.

Usage of (?expr as ?var) alias is obligatory.

Similarly to SQL, SPARQL allows computing aggregates over particular data groups and filter in them using **GROUP BY/HAVING** construct.

## Aggregation – Examples

*Listing 9 : Compute the number of inventions of each inventor.*

```
PREFIX : <http://example.org/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT (COUNT(?s) as ?count) ?i (GROUP_CONCAT(?s;separator=",") as ?inventors)
FROM :inventors
WHERE {
  ?s :invented ?i.
}
GROUP BY ?i
HAVING (COUNT(?s) > 1)
```

## Variable assignment

### Description

Variables can be assigned results of function (or aggregation function). The syntax is (expr **AS** ?v), where expr is an expression and ?v is the newly create variable not appearing before.

*Listing 10 : Compute the number of inventions of each inventor.*

```
PREFIX : <http://example.org/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT (COUNT(?s) AS ?count) ?invention
FROM :inventors
WHERE {
  ?s :invented ?i .
  ?i rdfs:label ?l
  BIND (concat("Invention:_",?l) AS ?invention)
}
GROUP BY ?i ?invention
```

## Distributed Queries

### Syntax and semantics

**syntax** ... **SERVICE** (**SILENT**) sparqlServiceURI { GP }

**semantics** this clause poses a sparql query described by graph pattern GP to a remote SPARQL endpoint *sparqlServiceURI*

Listing 11 : DBPedia service query

```
PREFIX : <http://example.org/>
PREFIX p: <http://dbpedia.org/property/>
PREFIX r: <http://dbpedia.org/resource/>
SELECT ?s ?p ?o ?i
WHERE {
  GRAPH :inventors { ?s :invented ?i. }
  OPTIONAL { SERVICE SILENT
    <http://dbpedia.org/sparql> {
      ?s ?p ?o
      FILTER( strstarts(str(?p),
        concat(str(p:), "death") ) ) } } }
```

Listing 12 : Local repo content

```
@prefix : <http://example.org/>
@prefix p: <http://dbpedia.org/property/>
@prefix r: <http://dbpedia.org/resource/>
:inventors {
  r:Thomas_Edison :invented :bulb.
  r:J_Cimrman :invented :bulb.
}
```

## Other Features

- **VALUES** – predefined variable binding specified in the tabular form
- **ORDER BY**, **LIMIT**, **OFFSET** – used analogously to SQL
- **FROM**, **FROM NAMED** – used to specify active default/named graphs for the query
- **SELECT DISTINCT** – removes duplicates from the results

## SPARQL SELECT/ASK results

**CSV** for **SELECT**; loses information about datatypes/languages of RDF terms

**TSV** for **SELECT**; is lossless

**XML**, **JSON** for **SELECT**, **ASK**; is lossless, supports additional information (e.g. columns identification through *link* attribute),



```
{
  "head": {
    "vars": [ "person", "name" ]
  },
  "results": {
    "bindings":
    [
      {
        "person": {
          "type": "uri",
          "value": "http://ex.com/p1" },
        "name": {
          "type": "literal",
          "value": "Smith" }
      },
      {
        "person": {
          "type": "uri",
          "value": "http://ex.com/p2" }
      }
    ]
  }
}
```

### Related Technologies

**SPIN** (SPARQL inference notation) – SPARQL rules encoded in RDF (<http://spinrdf.org/>)

**iSPARQL** – SPARQL visual query builder (<http://oat.openlinksw.com/isparql/>)

**SNORQL** – Web front-end for exploring SPARQL endpoints (<https://github.com/kurtjx/SNORQL>)

**SeRQL** – Sesame query language (alternative to SPARQL)

**SQWRL** (Semantic Query-Enhanced Web Rule Language) – query language based on SWRL (see next lecture), <http://protege.cim3.net/cgi-bin/wiki.pl?SQWRL>

### References