TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

KNOWLEDGE GRAPHS

Lecture 5: Advanced Features of SPARQL

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 28 Nov 2024

More recent versions of this slide deck might be available.
For the most current version of this course, see
https://iccl.inf.tu-dresden.de/web/Knowledge_Graphs/en

Review

SPARQL:

- . . . is the W3C-standard for querying RDF graphs
- . . . at its core relies on basic graph patterns (BGPs)
- . . . returns sequences or multi-sets of partial functions (“solutions”)

Wikidata:

- . . . is a large, free knowledge graph & open community
- . . . can be viewed as a document-centric or graph-based database
- . . . provides an RDF-mapping, linked-data exports, and the SPARQL-based Wikidata query service (WDQS)

In this lecture: many more SPARQL features

Property Paths

Paths for making connections

Knowledge graphs are about (indirect) connections – property paths are used to specify conditions on them:

Example 5.1: Find all descendants of Johann Sebastian Bach:

```
PREFIX eg: <http://example.org/>
SELECT ?descendant
WHERE { eg:JSBach eg:hasChild+ ?descendant . }
```

More complex paths are possible:

Example 5.2: Find all descendants of Johann Sebastian Bach in an RDF graph using no predicate `hasChild` but two predicates `hasFather` and `hasMother`:

```
PREFIX eg: <http://example.org/>
SELECT ?descendant
WHERE { eg:JSBach (^eg:hasFather|^eg:hasMother)+ ?descendant . }
```

The prefix `^` reverses the direction of edge traversal, and `|` expresses alternative.

Property path syntax

SPARQL supports the following property paths, where `iri` is a Turtle-style IRI (abbreviated or not) and `PP` is another property path:

Syntax	**Intuitive meaning**
`iri`	A path of length 1
`PP*`	A path consisting of 0 or more matches of the path `PP`
`PP+`	A path consisting of 1 or more matches of the path `PP`
`PP?`	A path consisting of 0 or 1 matches of the path `PP`
`^PP`	A path consisting of a match of `PP` in reverse order
$\texttt{PP}_1$ `/` $\texttt{PP}_2$	A path consisting of a match of $\texttt{PP}_1$, followed by a match of $\texttt{PP}_2$
$\texttt{PP}_1$ `|` $\texttt{PP}_2$	A path consisting of a match of $\texttt{PP}_1$ or by a match of $\texttt{PP}_2$
`!(`$\texttt{iri}_1$`|`$\cdots$`|`$\texttt{iri}_n$`)`	A length-1 path labelled by none of the given IRIs
`!(^`$\texttt{iri}_1$`|`$\cdots$`|^`$\texttt{iri}_n$`)`	A length-1 reverse path labelled by none of the given IRIs

Note that `!` cannot be applied to arbitrary property paths.

Property path syntax: precedence

- Parentheses can be used to control precedence.
- Negated property sets (`!`) must always be in parentheses, unless there is just one element
- The natural precedence of operations is: `!` $\succ$ */+/? $\succ$ `^` $\succ$ `/` $\succ$ `|`

Example 5.3: The property path `^!^eg:p1*/eg:p2?|eg:p3+` is interpreted as `(((^((!(^eg:p1))*))/(eg:p2?))|(eg:p3+))`, where we note:

- the interpretation of `!^eg:p1` is fixed and not subject to any precedence; it's simply a short form for the official syntax `!(^eg:p1)`,
- the meaning of `^(iri*)` is actually the same as the meaning of `(^iri)*`, so this precedence is inessential,
- the meaning of $\texttt{(iri}_1\texttt{/iri}_2\texttt{)|iri}_3$ is not the same as the meaning of $\texttt{iri}_1\texttt{/(iri}_2\texttt{|iri}_3\texttt{)}$.

Property path patterns

Definition 5.4: A property path pattern is a triple $\langle \mathtt{s}, \mathtt{p}, \mathtt{o} \rangle$, where $\mathtt{s}$ and $\mathtt{o}$ are arbitrary RDF terms, and $\mathtt{p}$ is property path.

Note: As for triple patterns, this is an abstract notion, which is syntactically represented by extending Turtle syntax to allow for property path expressions in predicate positions.

Example 5.5: Some property path patterns and their intuitive meaning:

1. `?x !(eg:p|eg:q)* ?y`: pairs of resources that are connected by a directed path of length ≥ 0 consisting of edges labelled neither `eg:p` nor `eg:q`
2. `?x eg:p/eg:p eg:o`: resources connected to `eg:o` by an `eg:p`-path of length 2; same as BGP `?x eg:p [eg:p eg:o]`
3. `?x (!eg:p|!^eg:q)* ?y`: pairs connected by a path of length ≥ 0 built from forward edges not labelled `eg:p` and reverse edges not labelled `eg:q`

Warning: SPARQL allows (3) to be written as `?x !(eg:p|^eg:q)* ?y`, which is confusing since `!(eg:p|^eg:q)` has more matches than `!(eg:p)`, whereas `!(eg:p|eg:q)` has less.

Property path semantics (1)

Recall the usual operations on languages $\mathbf{L}$, $\mathbf{L_1}$, and $\mathbf{L_2}$:

- $\mathbf{L_1} \circ \mathbf{L_2} = \{w_1 w_2 \mid w_1 \in \mathbf{L_1}, w_2 \in \mathbf{L_2}\}$
- $\mathbf{L}^0 = \{\varepsilon\}$ (the language with only the empty word ε)
- $\mathbf{L}^{i+1} = \mathbf{L}^i \circ \mathbf{L}$ and $\mathbf{L}^* = \bigcup_{i \geq 0} \mathbf{L}^i$

We recursively define a language **path**(PP) of words over (forward or reverse) predicates for each property path expression PP as follows:

- **path**(`iri`) = {`iri`} and **path**(`^iri`) = {`^iri`}
- **path**(PP_1`/`PP_2) = **path**(PP_1) $\circ$ **path**(PP_2)
- **path**(PP`*`) = **path**(PP)* and **path**(PP`+`) = **path**(PP) $\circ$ **path**(PP)*
- **path**(PP`?`) = $\{\varepsilon\} \cup$ **path**(PP)
- **path**(PP_1`|`PP_2) = **path**(PP_1) $\cup$ **path**(PP_2)
- **path**(`^`PP) = $\{\mathrm{inv}(p_\ell) \cdots \mathrm{inv}(p_1) \mid p_1 \cdots p_\ell \in$ **path**(PP)$\}$ where inv(`iri`) = `^iri` and inv(`^iri`) = `iri`
- **path**(`!(`iri_1`|`$\cdots$`|`iri_n`)`) = $\{$`iri` $\mid$ `iri` $\notin \{\mathrm{iri}_1, \ldots, \mathrm{iri}_n\}\}$
- **path**(`!(^`iri_1`|`$\cdots$`|^`iri_n`)`) = $\{$`^iri` $\mid$ `iri` $\notin \{\mathrm{iri}_1, \ldots, \mathrm{iri}_n\}\}$

Property path semantics (2)

Using **path**(PP), we can now define the solution multiset of a property path pattern:

Definition 5.6: Given an RDF graph G and a property path pattern $P = \langle s, \text{pp}, o \rangle$, a solution mapping μ is a solution to P over G if it is defined exactly on the variable names in P and there is a mapping σ from blank nodes to RDF terms such that G contains a path from $\mu(\sigma(s))$ to $\mu(\sigma(o))$ that is labelled by a word in **path**(pp), where a label of the form `^iri` refers to a reverse edge with label `iri`.

The cardinality of μ in the multiset of solutions is the number of distinct such mappings σ. The multiset of all these solutions is denoted $\text{eval}_G(P)$, where we omit G if clear from the context.

Note 1: We allow for empty paths here: they exist from any element to itself.

Note 2: We do not count the number of distinct paths: only existence is checked.

Note 3: This is actually wrong. SPARQL 1.1 sometimes counts some paths . . .

Counting paths

In general, counting paths is not feasible:

- If a graph has loops, there might be infinitely many distinct paths
- Even if we restrict to simple paths, the number of distinct paths grows exponentially

$\rightsquigarrow$ SPARQL 1.1 gave up most path counting, especially for * and +

However, SPARQL counts some paths nonetheless:

- Sequences are counted, e.g., `?s eg:p/eg:q ?o` has the same solution multiset as `?s eg:p [eg:q ?o]`
- Alternatives are counted, e.g., `?s eg:p|eg:q ?o` may have multiplicities of 1 or 2 for each result
- Negation sets are also counted, e.g., `?s !eg:p ?o` might have multiplicities > 1

Wrapping these path expressions into non-counted expressions "erases" the count:

Example 5.7: The property path pattern `?s (eg:p|eg:q) ?o` may have solutions with multiplicities 1 or 2, but the pattern `?s (eg:p|eg:q)? ?o` can only have multiplicity 1 for any solution.

Property paths in BGPs

SPARQL allows the use of property path patterns among triple patterns.

Example 5.8: Find all descendants of Bach that were composers:

```
PREFIX eg: <http://example.org/>
SELECT ?descendant
WHERE {
 eg:JSBach eg:hasChild+ ?descendant .
 ?descendant eg:occupation eg:composer .
}
```

The intuitive meaning of this should be clear.

Projection and Solution Set Modifiers

From patterns to queries

SELECT clauses

- specify the bindings that get returned (projection = removal of some bindings from results)
- may define additional results computed by functions
- may define additional results computed by aggregates (see later)

Example 5.9: Find cities and their population densities:

```
SELECT ?city (?population/?area AS ?populationDensity)
WHERE {
 ?city rdf:type eg:city ;
     eg:population ?population ;
     eg:areaInSqkm ?area .
}
```

Projection and Duplicates

Projection can increase the multiplicity of solutions

Definition 5.10: The projection of a solutions mapping μ to a set of variables V is the restriction of the partial function μ to variables in V. The projection of a solution sequence is the sequence of all projections of its solution mappings (order is preserved).

The cardinality of a solution mapping μ in a solution Ω is the sum of the cardinalities of all mappings $\nu \in \Omega$ that project to the same mapping μ.

Note: This definition also works if additional results are defined by functions or aggregates. Solution mappings are extended first by adding the bound variables, and then subjected to projection.

The keyword `DISTINCT` can be used after `SELECT` to remove duplicate solutions
(=to set multiplicity of any element in the result to 1);
resulting order: the first occurrence of each projected solution is kept

Solution set modifiers

SPARQL supports several expressions after the query's WHERE clause:

- **ORDER BY** defines the desired order of results
  - Can be followed by several expressions (separated by space)
  - May use order modifiers **ASC()** (default) or **DESC()**
- **LIMIT** defines a maximal number of results
- **OFFSET** specifies the index of the first result within the list of all results

Both **LIMIT** and **OFFSET** should only be used on explicitly ordered results

Example 5.11: In Wikidata, find the largest German cities, rank 6 to 15:

```
SELECT ?city ?population
WHERE {
 ?city wdt:P31 wd:Q515 ; # instance of city
       wdt:P17 wd:Q183 ; # country Germany
       wdt:P1082 ?population # get population
} ORDER BY DESC(?population) OFFSET 5 LIMIT 10
```

Aggregates

Grouping and aggregates

Aggregate functions compute values from multisets of solution mappings (rather than from individual mappings)

Grouping is used to split a multiset of solutions into several multisets based on some key that is computed for each solution

Example 5.12: In Wikidata, find the ten most common professions of people born in Dresden:

```
SELECT ?job (COUNT(?person) as ?count)
WHERE {
  ?person wdt:P19 wd:Q1731 ; # born in: Dresden
          wdt:P106 ?job . # occuptation: ?job
} GROUP BY ?job
ORDER BY DESC(?count) LIMIT 10
```

Note: we can select non-aggregate terms used for grouping (since they are the same across the whole group!).

SPARQL aggregate functions

SPARQL offers several aggregate functions:

- COUNT: count the sum of all multiplicities of solutions
- SUM: sum up numeric values
- AVG: compute the average of numeric values
- MIN/MAX: compute the minimum/maximum (over any type of term)
- SAMPLE: non-deterministically get one value from all values (no probability distribution implied)
- GROUP_CONCAT: concatenate string values into one large string (in any order)

All aggregate functions receive one expression as parameter, e.g., `SUM(?population)` or `MIN(year(?birthdate))`.

All aggregates optionally accept `DISTINCT` before the parameter to indicate that duplicates should be eliminated from the multiset of expression results before applying the aggregate.

HAVING

The keyword **HAVING** is used to specify a filter condition on mappings produced by aggregation (we will discuss filters in detail later):

Example 5.13: In Wikidata, find all professions of more than 100 people born in Dresden:

```
SELECT ?job (COUNT(?person) as ?count)
WHERE {
 ?person wdt:P19 wd:Q1731 ; # born in: Dresden
         wdt:P106 ?job . # occuptation: ?job
} GROUP BY ?job
HAVING (COUNT(?person) > 100)
```

Filters

Filters

Filters are SPARQL query expressions that can express many conditions that are not based on the RDF graph structure:

- Numeric and arithmetic comparisons
- Datatype-specific conditions (e.g., comparing the year of a date)
- String matching (sub-string comparison, regular expression matching, . . .)
- Type checks and language checks
- Logical combinations of conditions
- Check for non-existence of certain graph patterns
- . . .

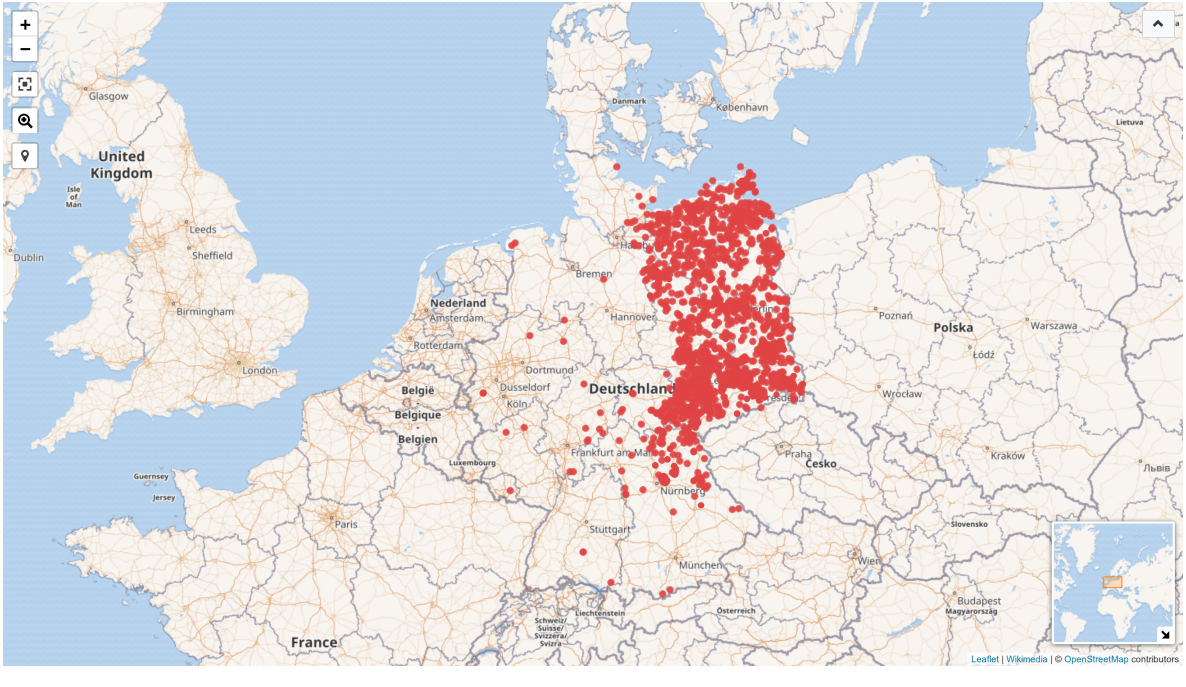
They are marked by the **`FILTER`** keyword.

Example

Example 5.14: From Wikidata, find out where in Germany towns have names ending in “-ow” or “-itz”:

```
SELECT ?item ?itemLabel ?coord WHERE {
 ?item wdt:P31/wdt:P279* wd:Q486972;
        # instances of (subclasses of) human settlement
     wdt:P17 wd:Q183; # country: Germany
     rdfs:label ?itemLabel; # get a label
     wdt:P625 ?coord # get coordinates
 FILTER (lang(?itemLabel) = "de") # label should be German ...
 FILTER regex (?itemLabel, "(ow|itz)$") # ... and end in -ow or -itz
}
```

Note: Filters are not Turtle syntax and don’t require `.` as separators.
But software will usually tolerate additional `.` before or after **FILTER** clauses.


Glasgow
United Kingdom
Isle of Man
Dublin
Leeds
Sheffield
Birmingham
London
Nederland
Amsterdam
Rotterdam
België
Belgique
Belgien
Luxembourg
Guernsey
Jersey
Paris
France
Danmark
København
Bremen
Hannover
Dortmund
Düsseldorf
Köln
Deutschland
Frankfurt am Main
Nürnberg
Stuttgart
München
Schweiz/ Suisse/ Svizzera/ Svizra
Liechtenstein
Österreich
Lietuva
Poznań
Polska
Warszawa
Łódź
Wrocław
Praha
Česko
Kraków
Львів
Slovensko
Wien
Budapest
Magyarország
Leaflet | Wikimedia | © OpenStreetMap contributors

How filters work

Rule 1: Filters cannot produce new answers or bind unbound variables.

- They just “filter” given answers by eliminating solutions that don’t satisfy a condition
- Answers are eliminated, never added
- Filter conditions only make sense on variables that occur in the pattern

Rule 2: The position of filters within a pattern is not relevant

- The filter condition always refers to answers to the complete pattern (not to parts)
- The relative order of several filters does not change the final outcome
- Implementations will optimise order (apply selective filters as early as possible)

Available filter conditions (1)

SPARQL supports many different filter conditions.

Comparison operators

- The familiar =, `!=`, <, >, <=, and >= are all supported
- Comparison of order are specific to the datatype of the element (the order on dates is different from the order on numbers etc.)
- = and `!=` compare two values (from the value space), not just syntactic forms
- Not all pairs of resources of distinct type might be comparable ($\leadsto$ error)

RDF-specific operators

- **`isIRI`**, **`isBlank`**, **`isLiteral`** test type of RDF term
- **`isNumeric`** checks if a term is a literal of a number type
- **`bound`** checks if a variable is bound to any term at all
- **`sameTerm`** checks if two terms are the same (not just equal-valued)

Available filter conditions (2)

SPARQL supports many different filter conditions.

String operators

- **`StrStarts`**, **`StrEnds`**, **`Contains`** test if string starts with/ends with/contains another
- **`RegEx`** checks if a string matches a regular expression
- **`langMatches`** checks if a string is a language code from given range of languages

Boolean operators

- Conditions can be combined using **`&&`** (and), **`||`** (or), and **`!`** (not)
- Parentheses can be used to group conditions

Functions in filters

Besides comparing constant terms and the values of given variables, filters can also include other function terms that compute results.

Arithmetic functions

The usual `+`, `*`, `-` (unary and binary), `/` are available, as well as **`abs`**, **`ceil`**, **`floor`**, **`round`**

String functions

These include **`SubStr`**, **`SubLen`**, **`StrBefore`**, **`StrAfter`**, **`Concat`**, **`Replace`**, and others

RDF term functions

Extract part of a term (**`datatype`**, **`lang`**); convert terms to other kinds of terms (**`str`**, **`iri`**, **`bnode`**, **`strDt`**, **`strLang`**, ...)

Date/time functions

Extract parts of a date: **`year`**, **`month`**, **`day`**, **`hour`**, ...

Logical functions

`IF` evaluates a term conditionally; **`COALESCE`** returns from a list of expressions the value of the first that evaluates without error; **`IN`** and **`NOTIN`** check membership of a term in a list

NOT EXISTS

SPARQL supports testing for absence of patterns in a graph using **NOT EXISTS**:

Example 5.15: From Wikidata, find out how many (known) living people are born in Dresden:

```
SELECT (COUNT(*) as ?count) WHERE {
  ?person wdt:P19 wd:Q1731 . # born in Dresden
  FILTER NOT EXISTS { ?person wdt:P570 [] } # no date of death
}
```

Any SPARQL query pattern can be used inside this filter.
This provides a form of negation in queries.

Variables in the pattern have a special meaning:

- variables bound in the filtered answer (for the surrounding pattern) are interpreted as in the answer
- unbound variables are interpreted as actual variables of the test query

Errors and effective boolean values

Observation:

- Many filter operations and functions only make sense for certain types of terms (e.g., `year` requires a date).
- RDF allows almost all kinds of terms in almost all positions.

$\rightsquigarrow$ variables might be bound to terms for which a filter makes no sense

Solution:

- Filter operations and functions might return “error” as a special value
- SPARQL defines how errors propagate
  Example: “true || error = true” but “true && error = error”
- Filters and boolean functions may use non-boolean inputs: in this case they assume their effective boolean value (EBV) as defined in the specification
  Example: numbers equivalent to 0 have EBV “false”, other numbers have EBV “true”
  Example: empty strings have EBV “false”, other strings have EBV “true”
  Example: errors have EBV “false”

Groups, Union, Minus, Optional, Subqueries

Groups

So far, all of our queries had a single pattern consisting of

- triple patterns
- property path patterns
- filters

When introducing further features, we will often have to group them:
this is done with braces `{ ... }`

Terminology: A query part within braces is called a group graph pattern in SPARQL.

We were already using group graph patterns in all queries: the part after **`WHERE`** is one

Semantically, results of juxtaposed group graph patterns are combined using Join.

Union

The **UNION** operator allows us to obtain the union of the results of two group graph patterns.

Example 5.16: In Wikidata, find everybody who is a composer by occupation or who has composed something:

```
SELECT ?person
WHERE {
  { ?person wdt:P106 wd:Q36834 } # ?person occupation: composer
  UNION
  { ?music wdt:P86 ?person } # ?music composer: ?person
}
```

UNION produces the union of results and adds up multiplicities

⇝ using **DISTINCT** might be necessary

Minus

The **MINUS** operator allows us to remove the results of one group graph pattern from the results of another.

Example 5.17: In Wikidata, find living people who are composers by occupation:

```
SELECT ?person
WHERE {
  { ?person wdt:P106 wd:Q36834 } # ?person occupation: composer
  MINUS
  { ?person wdt:P570 [] } # ?person date of death: some value
}
```

Similar results can often be achieved with **FILTER NOT EXISTS**, but the two are used differently:

MINUS and **FILTER NOT EXISTS** behave differently, e.g., when applied to group graph patterns that do not share any variables.

Optional

The **OPTIONAL** operator is used to extend solution mappings with additional, optional information.

Example 5.18: In Wikidata, find composers, and, optionally, their spouses:

```
SELECT ?person ?spouse
WHERE {
  ?person wdt:P106 wd:Q36834 # ?person occupation: composer
  OPTIONAL { ?person wdt:P26 ?spouse } # ?person spouse: ?spouse
}
```

Solutions for queries with **OPTIONAL** may leave some query variables unbound (people without spouses in the example).

Note: Like **FILTER**, **OPTIONAL** patterns are used inside one group graph pattern, together with triple patterns etc.

Subqueries

Subqueries are used to use results of other queries within queries, typically to achieve results that cannot be accomplished using other patterns.

Example 5.19: In Wikidata, find universities located in one of the 15 largest German cities:

```
SELECT DISTINCT ?university ?city
WHERE {
  { SELECT DISTINCT ?city ?population
    WHERE { ?city wdt:P31/wdt:P279* wd:Q515 ; # instance of: city
                  wdt:P17 wd:Q183 ; # country: Germany
                  wdt:P1082 ?population . # population: ?population
    } ORDER BY DESC(?population) LIMIT 15 # get top 15 by ?population
  }
  ?university wdt:P31/wdt:P279* wd:Q3918 ; # instance of: university
               wdt:P131+ ?city . # located in+: ?city
}
```

Interpretation of subqueries

The result multiset of the subquery is simply used like the result of any other (sub) group graph pattern.

Notes:

- The order of results from subqueries is not conveyed to the enclosing query (subqueries return multisets, not sequences).
- The use of **`ORDER BY`** is still meaningful to select top-k results by some ordering.
- Only selected variable names are part of the subquery result; other variables might be hidden from the enclosing query

Values and Bind

Defining own values

It is often useful to add bindings to results that do not come directly from the database:

- Predefine batches of (tuples of) constants $\rightsquigarrow$ **`VALUES`**
- Define derived values by applying functions to query results $\rightsquigarrow$ **`BIND`**

Both constructs behave slightly differently.

Values

VALUES is used to inject pre-defined result multisets into the query evaluation.

Example 5.20: In Wikidata, find people who are composers, or musicians, or who play some instrument:

```
SELECT DISTINCT ?item
WHERE {
  VALUES (?predicate ?value) { # define values for two variables
    ( wdt:P106 wd:Q36834 ) # occupation / composer
    ( wdt:P106 wd:Q639669 ) # occupation / musician
    ( wdt:P1303 UNDEF ) # instrument played / any
  }
  ?item ?predicate ?value
}
```

The **VALUES** expression defines three solution mappings, two of which are defined for variable names `predicate` and `value`, and one defined for `predicate` only.

Note: One may leave away the (...) if values are given for just one variable.

Interpretation and usage of **VALUES**

VALUES behaves just like a subquery with the specified result.

- As with subqueries, order does not matter.
- The special value `UNDEF` is used to signify that a variable should be unbound for a solution mapping
- Otherwise, only IRIs or literals can be used in **VALUES** – especially no functions

In practice, the most important use of **VALUES** is to encode batch queries that ask for many possible options in a single query. Using this to ask about, say, 100 possible values in one query is much more efficient than sending 100 small queries or using nested **UNION** with 100 possibilities.

Bind

BIND is used to assign a computed value to a variable.

Example 5.21: Find cities and their population densities:

```
SELECT ?city ?populationDensity
WHERE {
 ?city rdf:type eg:city ;
     eg:population ?population ;
     eg:areaInSqkm ?area .
 BIND (?population/?area AS ?populationDensity)
}
```

BIND can be used instead of expression assignments with **AS** in **SELECT**

However, variables assigned with **BIND** can already be used in the query pattern, but not before they were assigned.

Assignments of constants to variables are better realised with **VALUES**, which can be used before or after other patterns using the variable.

Summary

Property Path Patterns are used to describe (arbitrarily long) paths in graphs

Filters can express many conditions to eliminate some of the query results

Solution set modifiers define standard operations on result (multi)sets

Important SPARQL query operators are **`UNION`**, **`MINUS`**, **`OPTIONAL`**, **`BIND`**, and **`VALUES`**

Aggregates are used to obtain answers that combine several solutions.

What's next?

- Algebra operations for defining the SPARQL semantics
- SPARQL complexity and implementation
- Expressive limits of SPARQL