TECHNISCHE UNIVERSITÄT DRESDEN

KNOWLEDGE GRAPHS

Lecture 7: Advanced Features of SPARQL (2)

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 27th Nov 2018

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

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

Example 7.1: 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 7.2: 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 set of all projections of its solution mappings, ordered by the first occurrence of each projected solution mapping.

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)

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

Example 7.3: 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 ; # county Germany
        wdt:P1082 ?population # get population
} ORDER BY DESC(?population) OFFSET 5 LIMIT 10
```

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 7.4: 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

$\rightsquigarrow$ using **DISTINCT** might be necessary

Semantics of SPARQL queries

SPARQL query features are defined by corresponding query algebra operations that produce results (i.e., multisets of solution mappings).

We already encountered some such operations:

- eval_G produced results for BGPs and property path patterns
- Join computed the natural join of two results

Semantics of SPARQL queries

SPARQL query features are defined by corresponding query algebra operations that produce results (i.e., multisets of solution mappings).

We already encountered some such operations:

- eval_G produced results for BGPs and property path patterns
- Join computed the natural join of two results

We omitted the according operation for **`FILTER`** so far. It is simple; we just need to take into account that the meaning of some filter expressions (e.g., **`NOT EXISTS`**) depends on the given RDF graph:

Definition 7.5: Given a filter expression φ, a multiset M of solution mappings, and an RDF graph G, we define the multiset

$$\text{Filter}(\varphi, M, G) = \{\mu \mid \mu \in M \text{ and } \varphi \text{ evaluates to true for } \mu \text{ (over } G)\}$$

with the cardinality of a solution mapping μ defined as $\text{card}_{\text{Filter}(\varphi,M,G)}(\mu) = \text{card}_M(\mu)$.

Semantics of **UNION**

The semantics of **UNION** is defined by the operation Union(M_1, M_2) that computes the union of two multisets M_1 and M_2 of solution mappings:

Definition 7.6: Given multisets M_1 and M_2 of solution mappings, we define the multiset

$$\text{Union}(M_1, M_2) = \{\mu \mid \mu \in M_1 \text{ or } \mu \in M_2\}$$

with the cardinality of a solution mapping μ defined as

$$\text{card}_{\text{Union}(M_1,M_2)}(\mu) = \text{card}_{M_1}(\mu) + \text{card}_{M_2}(\mu).$$

Minus

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

Example 7.7: 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 a group graph patterns that do not share any variables.

Semantics of **MINUS**

The semantics of **MINUS** is defined by the operation Minus(M_1, M_2) that computes the set difference of two results M_1 and M_2:

Definition 7.8: Given multisets M_1 and M_2 of solution mappings, we define the multiset

$$\text{Minus}(M_1, M_2) = \{\mu \mid \mu \in M_1 \text{ and for all } \mu' \in M_2 : \mu \text{ and } \mu' \text{ are not compatible or have disjoint domains: } \text{dom}(\mu) \cap \text{dom}(\mu') = \emptyset\}$$

with the cardinality of a mapping μ defined as $\text{card}_{\text{Minus}(M_1,M_2)}(\mu) = \text{card}_{M_1}(\mu)$.

Recall: mappings μ_1 and μ_2 are compatible if $\mu_1(x) = \mu_2(x)$ for all variable names $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$

Note: Minus(M_1, M_2) does not depend on cardinalities of mappings in M_2.

Optional

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

Example 7.9: 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.

Optional and filters

What does the following query mean?

Example 7.10:

```
SELECT ?person ?spouse
WHERE {
 ?person wdt:P106 wd:Q36834 ; # ?person occupation: composer
         wdt:P569 ?bd . # ?person date of birth: ?bd
 OPTIONAL {
   ?person wdt:P26 ?spouse . # ?person spouse: ?spouse
   ?spouse wdt:P569 ?bd2 .    # ?spouse date of birth: ?bd2
   FILTER (year(?bd)=year(?bd2)) # born in same year
 }
}
```

Optional and filters

What does the following query mean?

Example 7.10:

```
SELECT ?person ?spouse
WHERE {
 ?person wdt:P106 wd:Q36834 ; # ?person occupation: composer
         wdt:P569 ?bd . # ?person date of birth: ?bd
 OPTIONAL {
   ?person wdt:P26 ?spouse . # ?person spouse: ?spouse
   ?spouse wdt:P569 ?bd2 .    # ?spouse date of birth: ?bd2
   FILTER (year(?bd)=year(?bd2)) # born in same year
 }
}
```

SPARQL: “Composers, and, optionally, their spouses that were born in the same year.”

Semantics of `OPTIONAL`

The semantics of **`OPTIONAL`** is defined by the operation LeftJoin(M_1, M_2, φ, G) that augments solutions in M_1 with compatible solutions in M_2 if this combination satisfies the filter condition φ (w.r.t. graph G):

Definition 7.11: Given multisets M_1 and M_2 of solution mappings, a filter expression φ, and an RDF graph G, we define the multiset

$$\begin{aligned}\text{LeftJoin}(M_1, M_2, \varphi, G) = \text{Filter}(\varphi, \text{Join}(M_1, M_2), G) \cup \\ \{\mu_1 \in M_1 \mid \text{ for all } \mu_2 \in M_2 : \mu_1 \text{ incompatible } \mu_2 \text{ or} \\ \varphi \text{ evaluates to false on } \mu_1 \uplus \mu_2 \text{ (over } G)\}\end{aligned}$$

with the cardinality of each mapping μ being its cardinality
in Filter$(\varphi, \text{Join}(M_1, M_2), G)$ (in case $\mu \in \text{Filter}(\varphi, \text{Join}(M_1, M_2), G)$)
or in M_1 (in case $\mu \notin \text{Filter}(\varphi, \text{Join}(M_1, M_2), G)$).
Note that only one of the two cases can occur.

Recall: mappings μ_1 and μ_2 are compatible if $\mu_1(x) = \mu_2(x)$ for all variable names $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$

Subqueries

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

Example 7.12: 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
}
```

Semantics of subqueries

The semantics of subqueries does not require any special operator: 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 $\leadsto$ **`VALUES`**
- Define derived values by applying functions to query results $\leadsto$ **`BIND`**

Both constructs behave slightly differently.

Values

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

Example 7.13: 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.

Semantics 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 7.14: 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.

Semantics of **BIND**

The semantics of **BIND** is defined by the operation Extend(M, v, φ) that computes the extension of solution mappings in M by assigning the output of expression φ to variable name v.

Definition 7.15: Consider a variable name v and an expression φ. Given a solution mapping μ such that $v \notin \text{dom}(\mu)$, we define an extended mapping

$$\text{Extend}(\mu, v, \varphi) = \begin{cases} \mu \cup \{v \mapsto \text{eval}(\mu(\varphi))\} & \text{if eval}(\mu(\varphi)) \text{ is not “error”} \\ \mu & \text{if eval}(\mu(\varphi)) \text{ is “error”} \end{cases}$$

Given a multiset M of solution mappings, we define $\text{Extend}(M, v, \varphi) = \{\text{Extend}(\mu, v, \varphi) \mid \mu \in M\}$, where the cardinalities of extended mappings are the same as in M.

Notation: eval$(\mu(\varphi))$ denotes evaluation of the expression obtained from φ by replacing variables by their values in μ.

Summary: SPARQL algebra

We have already encountered a number of operators for extending results:

- Join(M_1, M_2): join compatible mappings from M_1 and M_2
- Filter(φ, M, G): remove from multiset M all mappings for which φ does not evaluate to EBV "true"
- Union(M_1, M_2): compute the union of mappings from multisets M_1 and M_2
- Minus(M_1, M_2): remove from multiset M_1 all mappings compatible with a non-empty mapping in M_2
- LeftJoin(M_1, M_2, φ, G): extend mappings from M_1 by compatible mappings from M_2 when filter condition is satisfied; keep remaining mappings from M_1 unchanged
- Extend(M, v, φ): extend all mappings from M by assigning v the value of φ.

SPARQL also defines operators for solution set modifiers, which work on lists of mappings ("ordered multisets"):

- OrderBy(L, condition): sort list by a condition
- Slice(L, start, length): apply limit and offset modifiers

Further operators exist, e.g., Distinct(L).

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 7.16: 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 produces by aggregation:

Example 7.17: 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)
```

Semantics of grouping

The semantics of **`GROUP BY`** is defined by the operation Group(Φ, M) that computes a mapping from keys (that we group by) to multisets (that are the groups of solution mappings).

Definition 7.18: Consider a list of expressions $\Phi = \langle \varphi_1, \ldots, \varphi_n \rangle$. Given a solution mapping μ, we define $\Phi(\mu)$ as the list $\langle \varphi_1(\mu), \ldots, \varphi_n(\mu) \rangle$ of values obtained by evaluating expressions for the bindings of μ.

Given a multiset M of solution mappings, we define

$$\text{Group}(\Phi, M) = \left\{ \Phi(\mu) \mapsto \{\mu' \in M \mid \Phi(\mu') = \Phi(\mu)\} \mid \mu \in M \right\}$$

where the cardinality of each solution within the sub-multisets is the same as its cardinality in M.

Note: We can group by multiple expressions, hence the list Φ rather than a single expression only (example: **`GROUP BY`** `?occupation year(?date)` would group by two expressions, where one is derived using a function)

Semantics of aggregate functions

Results that include aggregate function values are computed as follows:

- An aggregate function takes as input a mapping of the form $\{k_1 \mapsto M_1, \ldots, k_\ell \mapsto M_\ell\}$ from keys k_i to mulitsets M_i and produces a new mapping $\{k_1 \mapsto v_1, \ldots, k_\ell \mapsto v_\ell\}$ from keys to values.
- If several aggregates are selected in the query, they are joined by combining value assignments for the same key into a single solution mapping.

The formal definition in SPARQL is rather general (hence more complicated) to allow for extension points where tools can add support for more complex aggregates.

Semantics of aggregate functions

Results that include aggregate function values are computed as follows:

- An aggregate function takes as input a mapping of the form $\{k_1 \mapsto M_1, \ldots, k_\ell \mapsto M_\ell\}$ from keys k_i to mulitsets M_i and produces a new mapping $\{k_1 \mapsto v_1, \ldots, k_\ell \mapsto v_\ell\}$ from keys to values.
- If several aggregates are selected in the query, they are joined by combining value assignments for the same key into a single solution mapping.

The formal definition in SPARQL is rather general (hence more complicated) to allow for extension points where tools can add support for more complex aggregates.

Example 7.19: In Wikidata, find the most common professions of people born in Dresden, together with average birth years of people with this job:

```
SELECT ?job (COUNT(?person) as ?count) (AVG(year(?bdate)) as ?aYear)
WHERE {
  ?person wdt:P19 wd:Q1731 ; # born in: Dresden
          wdt:P106 ?job ; # occupation: ?job
          wdt:P569 ?bdate . # date of birth: ?bdate
} GROUP BY ?job ORDER BY DESC(?count)
```

Summary

Solutions set modifiers define standard operations on result sets

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

The semantics of SPARQL is defined using algebraic operators

Aggregates are used to obtain answers that combine several solutions.

What's next?

- Further background on SPARQL complexity and semantics
- Other graph models and their query languages
- Other aspects of graph analysis and management