TECHNISCHE UNIVERSITÄT DRESDEN

KNOWLEDGE GRAPHS

Lecture 6: Advanced Features of SPARQL

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 20th 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

WDQS: current usage

SPARQL is widely used on Wikidata:

- >100M requests per month (3.8M per day) in 2018
- Many applications using SPARQL as API in the back-end (e.g., mobile apps)
- Useful for advanced content analysis and data journalism
- Also playing an important role in Wikidata editing and quality control

WDQS: current usage

SPARQL is widely used on Wikidata:

- >100M requests per month (3.8M per day) in 2018
- Many applications using SPARQL as API in the back-end (e.g., mobile apps)
- Useful for advanced content analysis and data journalism
- Also playing an important role in Wikidata editing and quality control

. . . while keeping up quality of service:

- 50% of queries answered in <40ms (95% in <440ms; 99% in <40s)
- Less than 0.05% of queries time out
- Service has never been down so far
- However: keeping up with updates is increasingly problematic, and the usual >60sec update target has often been missed in the second half of 2018

Query statistics from Malyshev et al., “Getting the Most out of Wikidata: Semantic Technology Usage in Wikipedia’s Knowledge Graph”, ISWC 2018

System setup

What is necessary to provide a public online service at this scale?

System setup

What is necessary to provide a public online service at this scale?

WDQS runs a typical master-slave setup with a primary storage server and several secondary query servers.

- Currently (2018), three commodity query servers are used for WDQS (+3 more for geographic backup)
- Wikidata's primary servers use a typical LAMP stack (Linux, Apache, MariaDB, PHP)
- BlazeGraph used as free and open-source graph database for WDQS
- Standard HTTP cache (Varnish) for speeding up answers
- Custom script used to monitor changes and update the database accordingly
- Query servers are independent; incoming requests distributed by load balancing (LVS)

All software used is free & open source

WDQS core components: `https://github.com/wikimedia/wikidata-query-rdf`

Property Paths

Paths for making connections

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

Example 6.1: Find all descendants of Johann Sebastian Bach:

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

Paths for making connections

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

Example 6.1: 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 6.2: 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
$\mathtt{PP}_1$ `/` $\mathtt{PP}_2$	A path consisting of a match of $\mathtt{PP}_1$, followed by a match of $\mathtt{PP}_2$
$\mathtt{PP}_1$ `|` $\mathtt{PP}_2$	A path consisting of a match of $\mathtt{PP}_1$ or by a match of $\mathtt{PP}_2$
$\mathtt{!(iri}_1\mathtt{\mid}\cdots\mathtt{\mid iri}_n\mathtt{)}$	A length-1 path labelled by none of the given IRIs
$\mathtt{!(\hat{}iri}_1\mathtt{\mid}\cdots\mathtt{\mid\hat{}iri}_n\mathtt{)}$	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: `!` $\succ$ `*/+/?` $\succ$ `ˆ` $\succ$ `/` $\succ$ `|`

Example 6.3: The property path `ˆ!ˆeg:p1*/eg:p2?|eg:p3+` is interpreted as `(((ˆ((!(ˆeg:p1))*))/(eg:p2?))|(eg:p3+))`, where we note:

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

Property path patterns

Definition 6.4: A property path pattern is a triple $\langle \mathrm{s}, \mathrm{p}, \mathrm{o} \rangle$, where s and o are arbitrary RDF terms, and 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.

Property path patterns

Definition 6.4: 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 6.5: 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 $\mathbf{L}^* = \bigcup_{i \geq 0} \mathbf{L}^i$

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**$(\mathtt{PP})$ of words over (forward or reverse) predicates for each property path expression $\mathtt{PP}$ as follows:

- **path**$(\mathtt{iri}) = \{\mathtt{iri}\}$ and **path**$(\hat{}\,\mathtt{iri}) = \{\hat{}\,\mathtt{iri}\}$
- **path**$(\mathtt{PP}_1/\mathtt{PP}_2)$ = **path**$(\mathtt{PP}_1) \circ$ **path**$(\mathtt{PP}_2)$
- **path**$(\mathtt{PP*})$ = **path**$(\mathtt{PP})^*$ and **path**$(\mathtt{PP+})$ = **path**$(\mathtt{PP}) \circ$ **path**$(\mathtt{PP})^*$
- **path**$(\mathtt{PP?}) = \{\varepsilon\} \cup$ **path**$(\mathtt{PP})$
- **path**$(\mathtt{PP}_1 \,|\, \mathtt{PP}_2)$ = **path**$(\mathtt{PP}_1) \cup$ **path**$(\mathtt{PP}_2)$
- **path**$(\hat{}\,\mathtt{PP}) = \{\mathsf{inv}(p_\ell)\cdots\mathsf{inv}(p_1) \mid p_1\cdots p_\ell \in$ **path**$(\mathtt{PP})\}$ where $\mathsf{inv}(\mathtt{iri}) = \hat{}\,\mathtt{iri}$ and $\mathsf{inv}(\hat{}\,\mathtt{iri}) = \mathtt{iri}$
- **path**$(\mathtt{!(iri}_1 | \cdots | \mathtt{iri}_n\mathtt{)}) = \{\mathtt{iri} \mid \mathtt{iri} \notin \{\mathtt{iri}_1, \ldots, \mathtt{iri}_n\}\}$
- **path**$(\mathtt{!(\hat{}iri}_1 | \cdots | \mathtt{\hat{}iri}_n\mathtt{)}) = \{\hat{}\,\mathtt{iri} \mid \mathtt{iri} \notin \{\mathtt{iri}_1, \ldots, \mathtt{iri}_n\}\}$

Property path semantics (2)

Using **path**$(\mathtt{PP})$, we can now define the solution multiset of a property path pattern:

Definition 6.6: 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.

Property path semantics (2)

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

Definition 6.6: 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

⇝ SPARQL 1.1 gave up most path counting, especially for * and +

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

$\leadsto$ 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

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

$\leadsto$ 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 6.7: 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 6.8: 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 .
}
```

Property paths in BGPs

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

Example 6.8: 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 semantics of these constructions is easy to define:

- If P_1 is a BGP and P_2 is a property path pattern without bnodes, then eval$(P_1 \cup P_2) =$ Join(eval(P_1), eval(P_2)).
- Further property path patterns can be included with further joins.
- Bnodes can be allowed by treating them like variables that are projected out later.

Filters

Filters

Filters are SPARQL query expressions for 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.

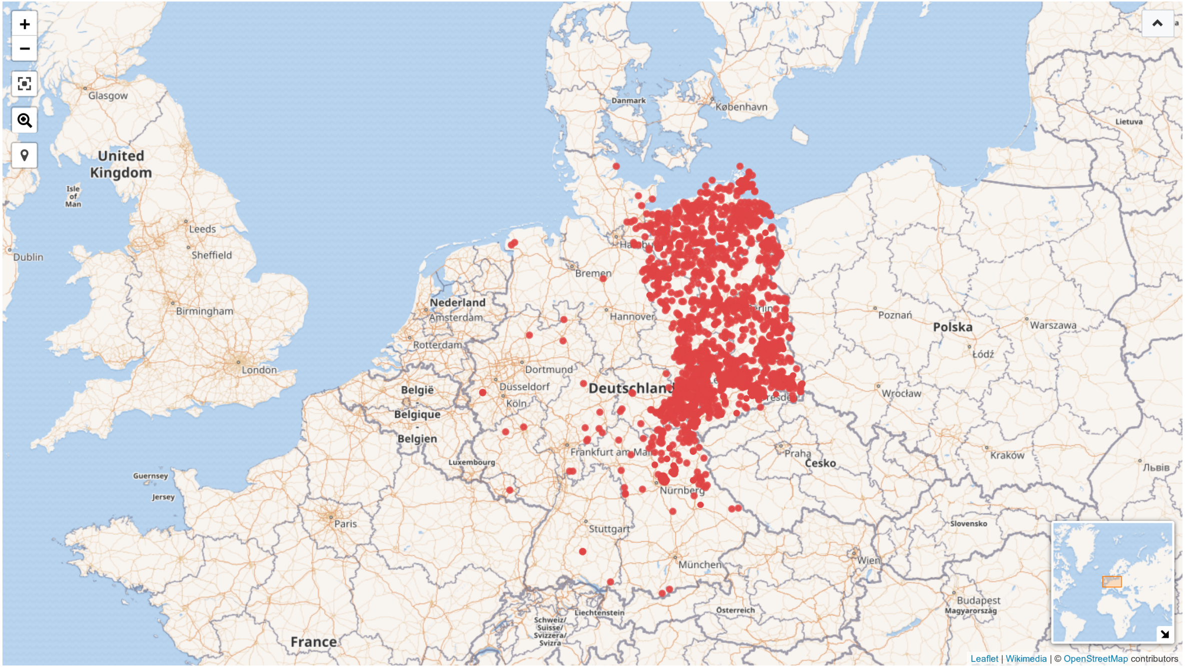
Example

Example 6.9: 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
Danmark
København
Lietuva
United Kingdom
Isle of Man
Leeds
Dublin
Sheffield
Hamburg
Bremen
Nederland
Amsterdam
Hannover
Berlin
Poznań
Polska
Warszawa
Birmingham
Rotterdam
Łódź
London
Dortmund
België
Düsseldorf
Deutschland
Wrocław
Köln
Dresden
Belgique
Belgien
Frankfurt am Main
Praha
Česko
Kraków
Luxembourg
Guernsey
Jersey
Львів
Nürnberg
Paris
Stuttgart
Slovensko
München
Wien
Liechtenstein
Österreich
Budapest
Magyarország
Schweiz/ Suisse/ Svizzera/ Svizra
France
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

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 to values (from the value space), not just syntactic forms
- Not all pairs of resources of distinct type might be comparable ($\leadsto$ error)

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 to values (from the value space), not just syntactic forms
- Not all pairs of resources of distinct type might be comparable (⇝ 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

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 if 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 6.10: From Wikidata, find out how many living people are know who 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.

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

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.

⇝ 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"

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 (next lecture)

Example 6.11: 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 6.12: 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 6.13: 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
```

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

Solutions set modifiers define standard operations on result sets

What's next?

- More SPARQL features: aggregates, subqueries, union, optional
- Further background on SPARQL complexity and semantics
- Other graph models and their query languages