TECHNISCHE UNIVERSITÄT DRESDEN

KNOWLEDGE GRAPHS

Lecture 6: SPARQL Semantics

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 19th Nov 2019

Review

SPARQL is a feature-rich query language:

- Basic graph patterns (conjunctions of triple patterns)
- Property path patterns
- Filters
- Union, Optional, Minus
- Subqueries, Values, Bind
- Solution set modifiers
- Aggregates

Review: Answers to BGPs

What is the result of a SPARQL query?

Definition 6.1: A solution mapping is a partial function μ from variable names to RDF terms. A solution sequence is a list of solution mappings.

Note: When no specific order is required, the solutions computed for a SPARQL query can be represented by a multiset (= "a set with repeated elements" = "an unordered list").

Definition 6.2: Given an RDF graph G and a BGP P, 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 in P to RDF terms, such that $\mu(\sigma(P)) \subseteq G$.

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

Note: Here, we write $\mu(\sigma(P))$ to denote the graph given by the triples in P after first replacing bnodes according to σ, and then replacing variables according to μ.

Understanding BGP Multiplicities (1)

```
G =  eg:Arrival eg:actorRole eg:aux1, eg:aux2 .
     eg:aux1 eg:actor eg:Adams  ; eg:character "Louise Banks" .
     eg:aux2 eg:actor eg:Renner ; eg:character "Ian Donnelly" .
     eg:Gravity eg:actorRole [ eg:actor eg:Bullock;
                               eg:character "Ryan Stone" ] .
```

BGP P_1 = `?film eg:actorRole ?ar . ?ar eg:actor ?person .` has solution multiset:

film	ar	person	cardinality
eg:Arrival	eg:aux1	eg:Adams	1
eg:Arrival	eg:aux2	eg:Renner	1
eg:Gravity	_:1	eg:Bullock	1

For example, for μ : film $\mapsto$ `eg:Arrival`, ar $\mapsto$ `eg:aux1`, person $\mapsto$ `eg:Adams`, there is exactly one mapping $\sigma : \emptyset \to$ RDF Terms (defined on the bnodes in P_1), such that $\mu(\sigma(P_1)) \subseteq G$.

Understanding BGP Multiplicities (2)

$P_1 =$ `?film eg:actorRole ?ar . ?ar eg:actor ?person .`

film	**ar**	**person**	cardinality
`eg:Arrival`	`eg:aux1`	`eg:Adams`	1
`eg:Arrival`	`eg:aux2`	`eg:Renner`	1
`eg:Gravity`	`_:1`	`eg:Bullock`	1

Understanding BGP Multiplicities (2)

$P_1 =$ `?film eg:actorRole ?ar . ?ar eg:actor ?person .`

film	**ar**	**person**	cardinality
eg:Arrival	eg:aux1	eg:Adams	1
eg:Arrival	eg:aux2	eg:Renner	1
eg:Gravity	_:1	eg:Bullock	1

$P_2 =$ `?film eg:actorRole [ eg:actor ?person ]`

Understanding BGP Multiplicities (2)

P_1 = `?film eg:actorRole ?ar . ?ar eg:actor ?person .`

film	**ar**	**person**	cardinality
`eg:Arrival`	`eg:aux1`	`eg:Adams`	1
`eg:Arrival`	`eg:aux2`	`eg:Renner`	1
`eg:Gravity`	`_:1`	`eg:Bullock`	1

P_2 = `?film eg:actorRole [ eg:actor ?person ]`

film	**person**	cardinality
`eg:Arrival`	`eg:Adams`	1
`eg:Arrival`	`eg:Renner`	1
`eg:Gravity`	`eg:Bullock`	1

Understanding BGP Multiplicities (2)

$P_1 =$ `?film eg:actorRole ?ar . ?ar eg:actor ?person .`

film	ar	person	cardinality
`eg:Arrival`	`eg:aux1`	`eg:Adams`	1
`eg:Arrival`	`eg:aux2`	`eg:Renner`	1
`eg:Gravity`	`_:1`	`eg:Bullock`	1

$P_2 =$ `?film eg:actorRole [ eg:actor ?person ]`

film	person	cardinality
`eg:Arrival`	`eg:Adams`	1
`eg:Arrival`	`eg:Renner`	1
`eg:Gravity`	`eg:Bullock`	1

$P_3 =$ `?film eg:actorRole [ eg:actor [] ]`

Understanding BGP Multiplicities (2)

$P_1 =$ `?film eg:actorRole ?ar . ?ar eg:actor ?person .`

film	ar	person	cardinality
eg:Arrival	eg:aux1	eg:Adams	1
eg:Arrival	eg:aux2	eg:Renner	1
eg:Gravity	_:1	eg:Bullock	1

$P_2 =$ `?film eg:actorRole [ eg:actor ?person ]`

film	person	cardinality
eg:Arrival	eg:Adams	1
eg:Arrival	eg:Renner	1
eg:Gravity	eg:Bullock	1

$P_3 =$ `?film eg:actorRole [ eg:actor [] ]`

film	cardinality
eg:Arrival	2
eg:Gravity	1

Review: Projection and Duplicates

Projection can increase the multiplicity of solutions

Definition 6.3: 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.

$\leadsto$ using blank nodes in patterns has the same effect as using variables that are projected away
(but bnode values cannot be used to compute aggregates or computed functions)

Finding BGP solutions using joins

To answer BGPs, real graph database retrieve solutions for triple patterns and combine them with joins.

Finding BGP solutions using joins

To answer BGPs, real graph database retrieve solutions for triple patterns and combine them with joins.

Definition 6.4: Two solution mappings μ_1 and μ_2 are compatible if $\mu_1(x) = \mu_2(x)$ for all variable names $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, where dom is the domain on which a (partial) function is defined. In this case, $\mu_1 \uplus \mu_2$ is the mapping defined as

$$\mu_1 \uplus \mu_2(x) = \begin{cases} \mu_1(x) & \text{if } x \in \text{dom}(\mu_1) \\ \mu_2(x) & \text{if } x \in \text{dom}(\mu_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Finding BGP solutions using joins

To answer BGPs, real graph database retrieve solutions for triple patterns and combine them with joins.

Definition 6.4: Two solution mappings μ_1 and μ_2 are compatible if $\mu_1(x) = \mu_2(x)$ for all variable names $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, where dom is the domain on which a (partial) function is defined. In this case, $\mu_1 \uplus \mu_2$ is the mapping defined as

$$\mu_1 \uplus \mu_2(x) = \begin{cases} \mu_1(x) & \text{if } x \in \text{dom}(\mu_1) \\ \mu_2(x) & \text{if } x \in \text{dom}(\mu_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 6.5: The join of two multisets Ω_1 and Ω_2 of solution mappings is the multiset $\text{Join}(\Omega_1, \Omega_2) = \{\mu_1 \uplus \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \text{ and } \mu_1 \text{ and } \mu_2 \text{ are compatible}\}$.

The multiplicity $\text{card}_\Omega(\mu)$ of each solution $\mu \in \Omega = \text{Join}(\Omega_1, \Omega_2)$ is given as $\text{card}_\Omega(\mu) = \sum_{\mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \uplus \mu_2 = \mu} \text{card}_{\Omega_1}(\mu_1) \times \text{card}_{\Omega_2}(\mu_2)$.

Finding BGP solutions using joins

Theorem 6.6: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then

$$\mathrm{BGP}_G(P) = \mathrm{Join}(\mathrm{BGP}_G(P_1), \mathrm{BGP}_G(P_2)).$$

So $\mathrm{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Finding BGP solutions using joins

Theorem 6.6: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then

$$\text{BGP}_G(P) = \text{Join}(\text{BGP}_G(P_1), \text{BGP}_G(P_2)).$$

So $\text{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Proof: Since P contains no bnodes, solutions are defined without considering mappings "σ" and the multiplicity of any solution will therefore be 1.

Finding BGP solutions using joins

Theorem 6.6: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then

$$\mathsf{BGP}_G(P) = \mathsf{Join}(\mathsf{BGP}_G(P_1), \mathsf{BGP}_G(P_2)).$$

So $\mathsf{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Proof: Since P contains no bnodes, solutions are defined without considering mappings "σ" and the multiplicity of any solution will therefore be 1.

"$\subseteq$" Consider $\mu \in \mathsf{BGP}_G(P)$.

- Let μ_i be the restriction of μ to variables in P_i $(i = 1, 2)$
- Then $\mu_i \in \mathsf{BGP}_G(P_i)$ and μ_1 and μ_2 are compatible
- Therefore $\mu_1 \uplus \mu_2 = \mu \in \mathsf{Join}(\mathsf{BGP}_G(P_1), \mathsf{BGP}_G(P_2))$

Finding BGP solutions using joins

Theorem 6.6: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then

$$\text{BGP}_G(P) = \text{Join}(\text{BGP}_G(P_1), \text{BGP}_G(P_2)).$$

So $\text{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Proof: Since P contains no bnodes, solutions are defined without considering mappings "σ" and the multiplicity of any solution will therefore be 1.

"$\subseteq$" Consider $\mu \in \text{BGP}_G(P)$.

- Let μ_i be the restriction of μ to variables in P_i $(i = 1, 2)$
- Then $\mu_i \in \text{BGP}_G(P_i)$ and μ_1 and μ_2 are compatible
- Therefore $\mu_1 \uplus \mu_2 = \mu \in \text{Join}(\text{BGP}_G(P_1), \text{BGP}_G(P_2))$

"$\supseteq$" Consider $\mu \in \text{Join}(\text{BGP}_G(P_1), \text{BGP}_G(P_2))$.

- Then there are compatible $\mu_i \in \text{BGP}_G(P_i)$ with $\mu_1 \uplus \mu_2 = \mu$
- By construction, $\mu_1(P_1) = \mu(P_1) \subseteq G$ and $\mu_2(P_2) = \mu(P_2) \subseteq G$
- Hence $\mu_1(P_1) \cup \mu_2(P_2) = \mu(P_1) \cup \mu(P_2) = \mu(P_1 \cup P_2) \subseteq G$, as claimed

Finding BGP solutions using joins

Theorem 6.6: Let G be an RDF graph, and let $P = P_1 \cup P_2$ be a bnode-free BGP that is a disjoint union of two BGPs P_1 and P_2. Then

$$\mathrm{BGP}_G(P) = \mathrm{Join}(\mathrm{BGP}_G(P_1), \mathrm{BGP}_G(P_2)).$$

So $\mathrm{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

Proof: Since P contains no bnodes, solutions are defined without considering mappings "σ" and the multiplicity of any solution will therefore be 1.

"$\subseteq$" Consider $\mu \in \mathrm{BGP}_G(P)$.

- Let μ_i be the restriction of μ to variables in P_i $(i = 1, 2)$
- Then $\mu_i \in \mathrm{BGP}_G(P_i)$ and μ_1 and μ_2 are compatible
- Therefore $\mu_1 \uplus \mu_2 = \mu \in \mathrm{Join}(\mathrm{BGP}_G(P_1), \mathrm{BGP}_G(P_2))$

"$\supseteq$" Consider $\mu \in \mathrm{Join}(\mathrm{BGP}_G(P_1), \mathrm{BGP}_G(P_2))$.

- Then there are compatible $\mu_i \in \mathrm{BGP}_G(P_i)$ with $\mu_1 \uplus \mu_2 = \mu$
- By construction, $\mu_1(P_1) = \mu(P_1) \subseteq G$ and $\mu_2(P_2) = \mu(P_2) \subseteq G$
- Hence $\mu_1(P_1) \cup \mu_2(P_2) = \mu(P_1) \cup \mu(P_2) = \mu(P_1 \cup P_2) \subseteq G$, as claimed

Finding BGP solutions . . . in practice

Theorem 6.6 does not work if the patterns contains blank nodes! (see exercise)

In practice, we can treat bnodes like variables that are projected away later on (leading to increased multiplicities).

Finding BGP solutions . . . in practice

Theorem 6.6 does not work if the patterns contains blank nodes! (see exercise)

In practice, we can treat bnodes like variables that are projected away later on (leading to increased multiplicities).

Real graph databases compute joins in highly optimised ways:

- Efficient data structures for finding compatible solutions to triple patterns (e.g., hash maps, tries, ordered lists, . . .)
- Query planners for optimising order of joins (goal: small intermediate results)
- Streaming joins: returning first results before join is complete
- Sometimes: multi-way joins (joining more than two triple patterns at once)

. . . but they still compute BGP solutions by joining partial solutions and hoping for an overall match

In the worst case, any known algorithm needs exponential time.

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:

- BGP_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:

- BGP_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 6.7: Given a filter expression φ, a multiset M of solution mappings, and an RDF graph G, we define the multiset

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

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

Semantics of `UNION`

The semantics of `UNION` is defined by the operation $\text{Union}(M_1, M_2)$ that computes the union of two multisets M_1 and M_2 of solution mappings:

Definition 6.8: 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

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

Semantics of `MINUS`

The semantics of `MINUS` is defined by the operation $\text{Minus}(M_1, M_2)$ that computes the set difference of two results M_1 and M_2:

Definition 6.9: Given multisets M_1 and M_2 of solution mappings, we define the multiset

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

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: $\text{Minus}(M_1, M_2)$ does not depend on cardinalities of mappings in M_2.

Semantics of `OPTIONAL`

The semantics of `OPTIONAL` is defined by the operation $\mathrm{LeftJoin}_G(M_1, M_2, \varphi)$ 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 6.10: Given multisets M_1 and M_2 of solution mappings, a filter expression φ, and an RDF graph G, we define the multiset

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

with the cardinality of each mapping μ being its cardinality in $\mathrm{Filter}_G(\varphi, \mathrm{Join}(M_1, M_2))$ (in case $\mu \in \mathrm{Filter}_G(\varphi, \mathrm{Join}(M_1, M_2))$) or in M_1 (in case $\mu \notin \mathrm{Filter}_G(\varphi, \mathrm{Join}(M_1, M_2))$).
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 $x \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$

Optional and filters

We defined LeftJoin to include filter conditions. Note the difference:

Example 6.11:

```
SELECT ?person ?spouse
WHERE {
 ?person eg:birthdate ?bd .
 OPTIONAL {
   ?person eg:spouse ?spouse .
   ?spouse eg:birthdate ?bd2 .
   FILTER (year(?bd)=year(?bd2))
 }
}
```

Example 6.12:

```
SELECT ?person ?spouse
WHERE {
 { ?person eg:birthdate ?bd .
   OPTIONAL {
     ?person eg:spouse ?spouse .
     ?spouse eg:birthdate ?bd2 .
   }
 }
 FILTER (year(?bd)=year(?bd2))
}
```

Optional and filters

We defined LeftJoin to include filter conditions. Note the difference:

Example 6.11:

```
SELECT ?person ?spouse
WHERE {
 ?person eg:birthdate ?bd .
 OPTIONAL {
   ?person eg:spouse ?spouse .
   ?spouse eg:birthdate ?bd2 .
   FILTER (year(?bd)=year(?bd2))
 }
}
```

"People with birthdate, and, optionally, their spouses born in the same year."

Example 6.12:

```
SELECT ?person ?spouse
WHERE {
 { ?person eg:birthdate ?bd .
   OPTIONAL {
     ?person eg:spouse ?spouse .
     ?spouse eg:birthdate ?bd2 .
   }
 }
 FILTER (year(?bd)=year(?bd2))
}
```

"Pairs of people with birthdate and spouses that were born in the same year."

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

Semantics of **`VALUES`** (review)

`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

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 6.13: 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 Extend(M, v, φ) $=$ $\{\text{Extend}(\mu, v, \varphi) \mid \mu \in M\}$, where the cardinalities of extended mappings are the same as in M.

Notation: eval$(\mu(\varphi))$ denotes the 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$_G(\varphi, M)$: 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$_G(M_1, M_2, \varphi)$: 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).

From query to algebra expression

A syntactic query expression can be transformed into an algebra expression, iteratively, inside out:

(1) Replace all basic graph patterns P with BGP(P)

(2) Replace all patterns of the form P `UNION` Q by Union(P, Q)

(3) Now select an innermost sequence S of expressions (all sub-patterns processed already)

   - Remove all FILTER expressions, and store them combined into a conjunction ψ
   - Initialise a result R to be the empty SPARQL expression Z
   - Process the remaining list of subexpressions SE iteratively
     - If SE is of the form `OPTIONAL` Filter(φ, A) then set $R :=$ LeftJoin(R, A, φ)
     - Else, if SE is of the form `OPTIONAL` A then set $R :=$ LeftJoin$(R, A, \mathtt{true})$
     - Else set $R :=$ Join(R, SE)
   - Finally, replace S by the expression Filter(ψ, R)

Review: 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 6.14: 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!).

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 6.15: 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 6.16: 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 ; # occuptation: ?job
         wdt:P569 ?bdate . # date of birth: ?bdate
} GROUP BY ?job ORDER BY DESC(?count)
```

Summary

SPARQL query results are multi-sets of answers (and lists if order was defined)

The semantics of SPARQL is defined using a variety of algebraic operators

SPARQL queries can be converted into nested expressions of operators that compute the result.

What's next?

- SPARQL complexity and implementation
- Expressive limits of SPARQL
- Other graph models and their query languages