Semantics of SPARQL

Werner Nutt

Acknowledgment

These slides are essentially identical with those by Sebastian Rudolph for his course on Semantic Web Technologies at TU Dresden

- Output Formats
- SPARQL Semantics
- Transformation of Queries into Algebra Expressions
- Evaluation of the SPARQL Algebra

- Output Formats
- SPARQL Semantics
- Transformation of Queries into Algebra Expressions
- Evaluation of the SPARQL Algebra

Output Format SELECT

So far all results have been tables (solution sequences):
Output format SELECT

Syntax:

- SELECT <VariableList>
- SELECT *

Advantage

- Simple sequential processing of the results

Disadvantage

- Structure/relationships is lost
  between the expressions in the result

Output Format CONSTRUCT

CONSTRUCT creates an RDF graph for the results

```
PREFIX ex: <http://example.org/>
CONSTRUCT { ?person  ex:mailbox  ?email .
                        ?person  ex:telephone  ?tel . }
WHERE {
      ?person  ex:email   ?email .
      ?person  ex:tel     ?tel .
}
```

Advantage

- Structured result data with relationships between the elements

Disadvantage

- Sequential processing of the results is harder
- No treatment of unbound variables (triples are omitted)

CONSTRUCT Templates with Blank Nodes

Data

```
@prefix  foaf:  <http://xmlns.com/foaf/0.1/> .
_:a         foaf:firstname     "Alice" ;
            foaf:surname       "Hacker" .
_:b         foaf:firstname     "Bob" ;
            foaf:surname       "Hacker" .
```

```
PREFIX foaf:     <http://xmlns.com/foaf/0.1/>
PREFIX vcard:    <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT {
 ?x       vcard:N               _:v .
 _:v      vcard:givenName  ?gname ;
          vcard:familyName ?fname
} WHERE {
 ?x       foaf:firstname     ?gname .
 ?x       foaf:surname       ?fname  }
```

CONSTRUCT Templates with Blank Nodes

Resulting RDF Graph

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
_:v1    vcard:N             _:x1 .
_:x1    vcard:givenName     "Alice" ;
        vcard:familyName    "Hacker" .
_:v2    vcard:N             _:x2 .
_:x2    vcard:givenName     "Bob" ;
        vcard:familyName    "Hacker" .
```

Further Output Formats: ASK & DESCRIBE

SPARQL supports two additional output formats:

- ASK only checks whether the query has at least one answer (true/ false result)
- DESCRIBE (informative) returns an RDF description for each resulting URI (application dependent)

Sample Query over DBpedia

```
PREFIX dbo:<http://dbpedia.org/ontology/>
PREFIX dbp:<http://dbpedia.org/property/>

DESCRIBE ?p WHERE {
 ?p      a                  dbo:Person .
 ?p      dbp:nationality    ?n .
 FILTER(REGEX(STR(?n),"ital","i"))
}
```

Output (just the beginning …)

```
@prefix dbpprop:    <http://dbpedia.org/property/> .
@prefix dbpedia:    <http://dbpedia.org/resource/> .
dbpedia:Emanuela_Da_Ros  dbpprop:birthPlace          dbpedia:Italy ;
          dbpprop:placeOfBirth         dbpedia:Italy .
@prefix dbpedia-owl:          <http://dbpedia.org/ontology/> .
dbpedia:Emanuela_Da_Ros  dbpedia-owl:birthPlace      dbpedia:Italy .
dbpedia:European_route_E66        dbpprop:countries dbpedia:Italy .
<http://dbpedia.org/resource/Fiori_(pasta)>     dbpprop:country   dbpedia:Italy ;

dbpedia:(
          dbpedia-owl:birthPlace       dbpedia:Italy .
dbpedia:Luigi_Cadorna       dbpprop:deathPlace          dbpedia:Italy ;
          dbpprop:placeOfBirth         dbpedia:Italy ;
          dbpprop:placeOfDeath         dbpedia:Italy ;
          dbpedia-owl:birthPlace       dbpedia:Italy ;
          dbpedia-owl:deathPlace       dbpedia:Italy .
dbpedia:Matteo_Abbate       dbpprop:birthPlace          dbpedia:Italy ;
          dbpprop:placeOfBirth         dbpedia:Italy ;
          dbpedia-owl:birthPlace       dbpedia:Italy .
```

The answer depends on the implementation and is not defined by the standard

- Output Formats
- SPARQL Semantics
- Transformation of Queries into Algebra Expressions
- Evaluation of the SPARQL Algebra

Semantics of Query Languages

So far only informal presentation of SPARQL features

- User: “Which answers can I expect for my query?”
- Developer: “Which behaviour is expected from my SPARQL implementation?”
- Marketing: “Is our product already conformant with the SPARQL standard?”

➔ Formal semantics should clarify these questions …

Logic-based Semantics

Semantics of formal logics:

- Model-theoretic semantics: Which interpretations do satisfy my given formulas?
- Proof-theoretic semantics: Which new formulas can be derived from my given formulas?
- …

Semantics of Programming Languages

- Axiomatic semantics:
  Which logical statements hold for my program?
- Operational semantics:
  What happens during the processing
  of my program?
- Denotational semantics:
  How can we describe the input/output function
  of the program in an abstract way?

Semantics of Query Languages (1)

Query Entailment

Logical view of queries and databases

- Query as description of allowed results
- Data as set of logical assumptions (axiom set/theory)
- Results as logical entailment

- OWL DL and RDF(S) as query languages
- Also logic programming view …

Semantics of Query Languages (2)

Query Algebra

Query as instruction for computing the results

- Queried data as input
- Results as output

➔ Relational algebra for SQL

➔ SPARQL algebra

See http://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra

- Output Formats
- SPARQL Semantics
- Transformation of Queries into Algebra Expressions
- Evaluation of the SPARQL Algebra

Translation into SPARQL Algebra

```
{ ?book ex:price ?price .
  FILTER (?price < 15)
  OPTIONAL { ?book ex:title ?title }
  { ?book ex:author ex:Shakespeare } UNION
  { ?book ex:author ex:Marlowe }
}
```

Semantics of a SPARQL query:

① Transformation of the query into an algebra expression

② Evaluation of the algebra expression

Translation into SPARQL Algebra

```
{ ?book ex:price ?price .
  FILTER (?price < 15)
  OPTIONAL { ?book ex:title ?title }
  { ?book ex:author ex:Shakespeare } UNION
  { ?book ex:author ex:Marlowe }
}
```

Attention: Filters apply to the whole group in which they occur

Translation into SPARQL Algebra

```
{ ?book ex:price ?price .
  OPTIONAL { ?book ex:title ?title }
  { ?book ex:author ex:Shakespeare } UNION
  { ?book ex:author ex:Marlowe }
  FILTER (?price < 15)
}
```

① Expand abbreviated IRIs

Translation into SPARQL Algebra

```
{ ?book <http://ex.org/price> ?price
  OPTIONAL { ?book <http://ex.org/title> ?title }
  { ?book <http://ex.org/author>
                <http://ex.org/Shakespeare> } UNION
  { ?book <http://ex.org/author>
                <http://ex.org/Marlowe> }
  FILTER (?price < 15)
}
```

Translation into SPARQL Algebra

```
{ ?book <http://ex.org/price> ?price
  OPTIONAL { ?book <http://ex.org/title> ?title }
  { ?book <http://ex.org/author>
                <http://ex.org/Shakespeare> } UNION
  { ?book <http://ex.org/author>
                <http://ex.org/Marlowe> }
  FILTER (?price < 15)
}
```

② Replace triple patterns (= basic graph patterns) with operator Bgp(·)

Translation into SPARQL Algebra

```
{ Bgp(?book <http://ex.org/price> ?price)
  OPTIONAL {Bgp(?book <http://ex.org/title> ?title)}
  {Bgp(?book <http://ex.org/author>
             <http://ex.org/Shakespeare>)} UNION
  {Bgp(?book <http://ex.org/author>
             <http://ex.org/Marlowe>)}
  FILTER (?price < 15)
}
```

Translation into SPARQL Algebra

```
{ Bgp(?book <http://ex.org/price> ?price)
  OPTIONAL {Bgp(?book <http://ex.org/title> ?title)}
  {Bgp(?book <http://ex.org/author>
             <http://ex.org/Shakespeare>)} UNION
  {Bgp(?book <http://ex.org/author>
             <http://ex.org/Marlowe>)}
  FILTER (?price < 15)
}
```

③ Introduce the LeftJoin(·) operator for optional parts

Translation into SPARQL Algebra

```
{LeftJoin(Bgp(?book <http://ex.org/price> ?price),
          Bgp(?book <http://ex.org/title> ?title),
          true)
  {Bgp(?book <http://ex.org/author>
             <http://ex.org/Shakespeare>)} UNION
  {Bgp(?book <http://ex.org/author>
             <http://ex.org/Marlowe>)}
  FILTER (?price < 15)
}
```

③ Introduce the LeftJoin(·) operator for optional parts

Note: LeftJoin(·, ·, ·) is a ternary operator

- 1st argument: mandatory part
- 2nd argument: Bgps of optional part
- 3rd argument: the filters of the optional group

Translation into SPARQL Algebra

```
{LeftJoin(Bgp(?book <http://ex.org/price> ?price),
          Bgp(?book <http://ex.org/title> ?title),
          true)
  {Bgp(?book <http://ex.org/author>
             <http://ex.org/Shakespeare>)} UNION
  {Bgp(?book <http://ex.org/author>
             <http://ex.org/Marlowe>)}
  FILTER (?price < 15)
}
```

④ Combine alternative graph patterns with the Union(·, ·) operator

➔ Refers to neighbouring patterns and has higher precedence than conjunction (left associative)

Translation into SPARQL Algebra

```
{LeftJoin(Bgp(?book <http://ex.org/price> ?price),
          Bgp(?book <http://ex.org/title> ?title),
          true)
 Union(Bgp(?book <http://ex.org/author>
                 http://ex.org/Shakespeare),
       Bgp(?book <http://ex.org/author>
                 http://ex.org/Marlowe))
  FILTER (?price < 15)
}
```

Translation into SPARQL Algebra

```
{LeftJoin(Bgp(?book <http://ex.org/price> ?price),
          Bgp(?book <http://ex.org/title> ?title),
          true)
 Union(Bgp(?book <http://ex.org/author>
                 http://ex.org/Shakespeare),
       Bgp(?book <http://ex.org/author>
                 http://ex.org/Marlowe))
  FILTER (?price < 15)
}
```

⑤ Apply Join(·,·) operator to join non-filter elements

Translation into SPARQL Algebra

```
{Join(
   LeftJoin(Bgp(?book <http://ex.org/price> ?price),
            Bgp(?book <http://ex.org/title> ?title),
            true) ,
   Union(Bgp(?book <http://ex.org/author>
                  http://ex.org/Shakespeare),
         Bgp(?book <http://ex.org/author>
                  http://ex.org/Marlowe)))
   FILTER (?price < 15)
}
```

Translation into SPARQL Algebra

```
{Join(
  LeftJoin(Bgp(?book <http://ex.org/price> ?price),
           Bgp(?book <http://ex.org/title> ?title),
           true),
  Union(Bgp(?book <http://ex.org/author>
                  http://ex.org/Shakespeare),
        Bgp(?book <http://ex.org/author>
                  http://ex.org/Marlowe)))
  FILTER (?price < 15)
}
```

⑥ Translate a group with filters with the Filter(·,·) operator

Translation into SPARQL Algebra

```
Filter(?price < 15 ,
   Join(
     LeftJoin(Bgp(?book <http://ex.org/price> ?price),
              Bgp(?book <http://ex.org/title> ?title),
              true) ,
     Union(Bgp(?book <http://ex.org/author>
                http://ex.org/Shakespeare),
            Bgp(?book <http://ex.org/author>
                http://ex.org/Marlowe))))
```

⑥ Translate a group with filters with the Filter(·,·) operator

Translation into SPARQL Algebra

```
Filter(?price < 15 ,
   Join(
     LeftJoin(Bgp(?book <http://ex.org/price> ?price),
              Bgp(?book <http://ex.org/title> ?title),
              true) ,
     Union(Bgp(?book <http://ex.org/author>
                 http://ex.org/Shakespeare),
           Bgp(?book <http://ex.org/author>
                 http://ex.org/Marlowe))))
```

Online translation tool:

http://sparql.org/query-validator.html

- Output Formats
- SPARQL Semantics
- Transformation of Queries into Algebra Expressions
- Evaluation of the SPARQL Algebra

Semantics of the SPARQL Algebra Operations

Now we have an algebra expression,
but what do the algebra operations mean?

Algebra Operator	Intuitive Semantics
Bgp(P)	match/evaluate pattern P
Join(M_1, M_2)	conjunctive join of solutions M_1 and M_2
Union(M_1, M_2)	union of solutions M_1 with M_2
LeftJoin(M_1, M_2, F)	optional join of M_1 with M_2 with filter constraint F (true if no filter given)
Filter(F, M)	filter solutions M with constraint F
Z	empty pattern (identity for join)

Only Bgp(·) matches or evaluates graph fragments …

Semantics of the SPARQL Algebra Operations

How can we define that more formally?

Output:

- “solution set” (formatting irrelevant)

Input:

- Queried (active) graph
- Partial results from previous evaluation steps
- Different parameters according to the operation

➔ How can we formally describe the “results”?

SPARQL Results

Intuition:

- Results are as for relational queries:
  tables of variable assignments

  Result:

  List of solutions (solution sequence)

  ➔ each solution corresponds to one table row

SPARQL Results

Formally:

A solution is a partial function (also called “mapping”) with

- Domain: relevant variables
- Range: IRIs ∪ blank nodes ∪ RDF literals

➔ Unbound variables are those that have no assigned value (partial function)

➔ Mappings are denoted by the greek letter μ

Evaluation of Basic Graph Patterns

Definition (Solution of a BGP)

Let P be a basic graph pattern.

A partial function/mapping μ is a solution for Bgp(P) over the queried (active) graph G if:

① the domain of μ is exactly the set of variables in P,

② there exists an assignment σ from blank nodes in P to IRIs, blank nodes, or RDF literals such that:

③ the RDF graph $\mu(\sigma(P))$ is a subgraph of G

Remarks on the Definition

- If there were only variables, we would only talk about μ.
- Since also the blank nodes need to be interpreted, there is also σ.
- It is first σ and then μ because we want
  - that σ only binds blank nodes in P,
  - not the blank nodes introduced by μ.
- The result of evaluating Bgp(P) over G is written

$$[[\text{Bgp}(P)]]_G$$

- The result is a multiset of solutions μ.
- The multiplicity of each solution μ corresponds to the number of different assignments σ

Multisets

Definition (Multi Set)

- A multiset over a set S is a function M that assigns to every element s of S
  - a natural number M(s) such that $M(s) \geq 0$ or $M(s) = \infty$ (infinity)
- M(s) is the multiplicity of s in M.

Alternative notation: {{ a, b, b }} corresponds to the multiset M over {a, b, c} with M(a) = 1, M(b) = 2, and M(c) = 0.

Solution Mappings: Example

```
ex:Werner ex:gives [
       a              ex:Lecture ;
       ex:hasTopic   "SPARQL" ] .
ex:Fariz ex:gives [
      a              ex:Lab ;
      ex:hasTopic   "Jena" ] .
```

Bgp(`?who ex:gives _:x . _:x ex:hasTopic ?what`)

Question:

- What are the σs and the μs?
- What are the solutions? And what is their multiplicity?

Hint: As a first step, write the data as a set of triples.

Solution Mappings: Exercise

```
ex:Fariz ex:gives [
       a              ex:Lab ;
       ex:hasTopic   ”RDF" ] .
ex:Fariz ex:gives [
      a              ex:Lab ;
      ex:hasTopic   "Jena" ] .
```

Bgp(`?who ex:gives _:x . _:x ex:hasTopic ?what`)

Question:

- What are the σs and the μs?
- What are the solutions? And what is their multiplicity?

Hint: As a first step, write the data as a set of triples.

Union of Solutions (1)

Definition (Compatibility)

Two solutions μ_1 and μ_2 are compatible if

$$\mu_1(x) = \mu_2(x) \text{ for all variables } x,$$

for which μ_1 and μ_2 are defined

Exercise: Find examples of μ_1 and μ_2 that are compatible/ not compatible

Union of Solutions (2)

Definition (Union)

The union of two compatible solutions μ_1 and μ_2 is a/the μ such that

- $\mu(x) = \mu_1(x)$ if x is in dom(μ_1)
- $\mu(x) = \mu_2(x)$ if x is in dom(μ_2)

Where does the compatibility play a role?

Evaluation of Join(·,·) (1)

To define the evaluation of a join expression Join(E_1,E_2) over a graph G we proceed in two steps:

① We define the join Join(M_1,M_2) of two multisets of mappings

② We define the evaluation $[[\text{Join}(E_1,E_2)]]_G$ of a join expression as the join of the evaluations $[[E_1]]_G$ and $[[E_2]]_G$ of the arguments

Evaluation of Join(·,·) (2)

For a mapping μ and multisets of mappings M_1, M_2 we define the set of join combinations of μ as

$$J(\mu) = \{ (\mu_1, \mu_2) \mid M_1(\mu_1) > 0, M_2(\mu_2) > 0, \mu_1 \text{ and } \mu_2 \text{ are compatible and } \mu = \mu_1 \cup \mu_2 \}$$

That is, $J(\mu)$ consists of all possible ways to obtain μ as a combination of mappings in M_1, M_2

Evaluation of Join(·,·) (3)

For multisets of mappings M_1, M_2 we define

$$\mathrm{Join}(M_1,M_2) := \{ (\mu,n) \mid n = \Sigma_{(\mu 1,\, \mu 2)\, \in J(\mu)} (M_1(\mu_1) * M_2(\mu_2)) \}$$

That is,

- Join(M_1,M_2) consists of all mappings μ that can be combined out of mappings in M_1 and M_2
- If μ can be combined out of μ_1 and μ_2, and
  μ_1 occurs n_1 times in M_1 and μ_2 occurs n_2 times in M_2,
  then this combination contributes $n_1 * n_2$
  to the multiplicity of μ in Join(M_1,M_2)

Evaluation of Join(·,·) (4)

Let E_1, E_2 be algebra expressions and let G be a graph.

Then we define

$$[[\text{Join}(E_1,E_2)]]_G := \text{Join}(\ [[E_1]]_G\ ,\ [[E_2]]_G\ )$$

In words: we evaluate the join of E_1 and E_2 by

- first evaluating E_1 and E_2 separately
- and then taking the join of the resulting multisets of mappings

Exercise for Join(·,·)

We consider algebra expressions E_1, E_2 and a graph G such that $[[E_1]]_G = M_1$ and $[[E_2]]_G = M_2$. We want to compute Join(E_1,E_2) over G. Suppose

$$M_1 = \{ ((\mu_1\text{: ?x} \rightarrow \text{ex:a, ?y} \rightarrow \text{ex:b}) ,2), \quad ((\mu_2\text{: ?x} \rightarrow \text{ex:a}, 1) \}$$

$$M_2 = \{ ((\mu_3\text{: ?y} \rightarrow \text{ex:b, ?z} \rightarrow \text{ex:c},3)\}$$

What is Join(M_1,M_2)?

I.e., which are the elements of the join?

And what is their multiplicity?

Evaluation of Union (1)

We first define the union of two multisets of assignments, and then the evaluation of a union expression.

Let M_1, M_2 be multisets of mappings. Then

$$\text{Union}(M_1,M_2) := \{ (\mu,n) \mid n = M_1(\mu) + M_2(\mu) > 0 \}$$

In words:

- the union contains the mappings that occur at least once in M_1 or M_2
- the multiplicity of a mapping μ in the union is the sum of the multiplicities in M_1 and M_2

Evaluation of Union (2)

Let E_1, E_2 be algebra expressions and let G be a graph.

Then we define

$$[[\text{Union}(E_1,E_2)]]_G := \text{Union}(\ [[E_1]]_G\ ,\ [[E_1]]_G\ )$$

In words: we evaluate the union of E_1 and E_2 by

- first evaluating E_1 and E_2 separately
- and then taking the union of the resulting multisets of mappings

Evaluation of Filter(·,·) (1)

To define the evaluation of a filter expression Filter(F,E) over a graph G we proceed in two steps:

① We define the filter operation Filter(F,M) of a filter condition F
and a multiset of mappings M

② We define the evaluation $[[\text{Filter}(F,E)]]_G$ of a filter expression as
the filter operation by F on the evaluation $[[E]]_G$

Evaluation of Filter(·) (2)

For a filter condition F and multiset of mappings M we define

$$\text{Filter}(F,M) := \{ (\mu,n) \mid M(\mu) = n > 0 \text{ and } \mu(F) = \text{T} \}$$

Here, $\mu(F)$ is the truth value (i.e., one of T, E, F) obtained from evaluating F with respect to μ.

The definition says that

① all mappings survive that satisfy the filter condition, and

② they survive with the multiplicity they had in M

Evaluation of Filter(·) (3)

Let E be an algebra expression, F a filter, and G a graph. The we define

$$[[\text{Filter}(F, E)]]_G := \text{Filter}(F, [[E]]_G)$$

In words,

- we first evaluate E and
- then apply the filter F

Evaluation of LeftJoin(·) (1)

Again, to define the evaluation of a left join expression LeftJoin(E_1,E_2,F) over a graph G we proceed in two steps:

① We define the left join LeftJoin(M_1,M_2,F) of two multisets of mappings and a filter condition

② We define the evaluation $[[\text{Join}(E_1, E_2, F)]]_G$ of a left join expression as the left join of the evaluations $[[E_1]]_G$ and $[[E_2]]_G$ of the arguments with respect to F

Evaluation of LeftJoin(·) (2)

Let M_1, M_2 be multisets of mappings and let F be a filter expression.

We define

$$\begin{aligned}\text{LeftJoin}(M_1, M_2, F) &:= \\ &\text{Filter}(F, \text{Join}(M_1, M_2)) \cup \\ &\{\, (\mu_1, M_1(\mu_1)) \mid \text{for all } \mu_2 \text{ with } M_2(\mu_2) > 0\text{: } \mu_1 \text{ and } \mu_2 \text{ are incompatible} \\ &\qquad \text{or } (\mu_1 \cup \mu_2)(F) \mathrel{!=} \mathrm{T} \,\}\end{aligned}$$

That is

- we join and filter as usual, and
- we keep those mappings from M_1 that
  - either do not find a match in M_2, or
  - for which none of the combinations with a match satisfies F

Evaluation of LeftJoin(·) (3)

Let E_1,E_2 be algebra expressions, F a filter, and G a graph. Then we define

$$[[\text{LeftJoin}(E_1,E_2,F)]]_G$$

$$:= \text{LeftJoin}([[E_1]]_G, [[E_2]]_G, F)$$

Example

```
@prefix ex:  <http://eg.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Hamlet            ex:author     ex:Shakespeare ;
                     ex:price      "10.50"^^xsd:decimal .
ex:Macbeth           ex:author     ex:Shakespeare .
ex:Tamburlaine       ex:author     ex:Marlowe ;
                     ex:price      "17"^^xsd:integer .
ex:DoctorFaustus     ex:author     ex:Marlowe ;
                     ex:price      "12"^^xsd:integer ; ;
         ex:title  "The Tragical History of Doctor Faustus" .
ex:RomeusJuliet      ex:author ex:Brooke ;
                     ex:price      ″12"^^xsd:integer .
```

```
{ ?book ex:price ?price .
  OPTIONAL { ?book ex:title ?title . }
  { ?book ex:author ex:Shakespeare . } UNION
  { ?book ex:author ex:Marlowe . }
  FILTER (?price < 15)
}
```

Example

```
@prefix ex:  <http://eg.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Hamlet            ex:author     ex:Shakespeare ;
                     ex:price      "10.50"^^xsd:decimal .
ex:Macbeth           ex:author     ex:Shakespeare .
ex:Tamburlaine       ex:author     ex:Marlowe ;
                     ex:price      "17"^^xsd:integer .
ex:DoctorFaustus     ex:author     ex:Marlowe ;
                     ex:price      "12"^^xsd:integer ; ;
        ex:title  "The Tragical History of Doctor Faustus" .
ex:RomeusJuliet      ex:author ex:Brooke ;
                     ex:price      ″12"^^xsd:integer .
```

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
           Bgp(?book ex:title ?title), true),
     Union(Bgp(?book ex:author ex:Shakespeare),
          Bgp(?book ex:author ex:Marlowe))))
```

Example Evaluation

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
           Bgp(?book ex:title ?title), true),
     Union(Bgp(?book ex:author ex:Shakespeare),
            Bgp(?book ex:author ex:Marlowe))))
```

book
ex:Macbeth
ex:Hamlet

Example Evaluation

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
           Bgp(?book ex:title ?title), true),
     Union(Bgp(?book ex:author ex:Shakespeare),
          Bgp(?book ex:author ex:Marlowe))))
```

book
ex:Tamburlaine
ex:DoctorFaustus

Example Evaluation

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
                Bgp(?book ex:title ?title), true),
       Union(Bgp(?book ex:author ex:Shakespeare),
             Bgp(?book ex:author ex:Marlowe))))
```

book
ex:Macbeth
ex:Hamlet
ex:Tamburlaine
ex:DoctorFaustus

Example Evaluation

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
             Bgp(?book ex:title ?title), true),
      Union(Bgp(?book ex:author ex:Shakespeare),
            Bgp(?book ex:author ex:Marlowe))))
```

book	price
ex:Hamlet	10.5
ex:Tamburlain	17
ex:DoctorFaustus	12
ex:RomeusJuliet	9

Example Evaluation

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
                Bgp(?book ex:title ?title), true),
       Union(Bgp(?book ex:author ex:Shakespeare),
             Bgp(?book ex:author ex:Marlowe))))
```

book	price
ex:Hamlet	10.5
ex:Tamburlain	17
ex:DoctorFaustus	12
ex:RomeusJuliet	9

book	title
ex:DoctorFaustus	"The Tragical History of Doctor Faustus"

Example Evaluation

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
                Bgp(?book ex:title ?title), true),
      Union(Bgp(?book ex:author ex:Shakespeare),
            Bgp(?book ex:author ex:Marlowe))))
```

book	price	title
ex:Hamlet	10.5	
ex:Tamburlain	17	
ex:DoctorFaustus	12	"The Tragical History of Doctor Faustus"
ex:RomeusJuliet	9	

Example Evaluation

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
           Bgp(?book ex:title ?title), true),
     Union(Bgp(?book ex:author ex:Shakespeare),
          Bgp(?book ex:author ex:Marlowe))))
```

book	**price**	**title**
ex:Hamlet	10.5	
ex:Tamburlain	17	
ex:DoctorFaustus	12	"The Tragical History of Doctor Faustus”

Example Evaluation

```
Filter(?price < 15,
  Join(LeftJoin(Bgp(?book ex:price ?price)
                Bgp(?book ex:title ?title), true),
       Union(Bgp(?book ex:author ex:Shakespeare),
             Bgp(?book ex:author ex:Marlowe))))
```

book	price	title
ex:Hamlet	10.5	
ex:DoctorFaustus	12	"The Tragical History of Doctor Faustus”

Formal Algebra Transformation

- During parsing of a query, a parse tree is constructed
- The parse tree contains expressions that correspond to the grammar
- For the transformation, we traverse the parse tree and recursively build the algebra expressions
- The query pattern is a `GroupGraphPattern` consisting of the following elements:
  - `TriplesBlock`
  - `Filter`
  - `OptionalGraphPattern`
  - `GroupOrUnionGraphPattern`
  - `GraphGraphPattern`

Part of the SPARQL Grammar

```
GroupGraphPattern            ::= '{' TriplesBlock?
                                   ( ( GraphPatternNotTriples
                                     | Filter ) '.' ? TriplesBlock? )*
                                 '}'

GraphPatternNotTriples       ::= OptionalGraphPattern
                                 | GroupOrUnionGraphPattern
                                 | GraphGraphPattern

OptionalGraphPattern         ::= 'OPTIONAL' GroupGraphPattern

GroupOrUnionGraphPattern     ::= GroupGraphPattern ( 'UNION'
                                   GroupGraphPattern )*

Filter                       ::= 'FILTER' Constraint
```

Transformation Algorithm to Algebra

Algorithm 1 translate(*G*)

Input: a query pattern *G*
Output: a SPARQL algebra expression *A*

```
if G is a Triplesblock then
    A := Bgp(G)
else if G is a GroupOrUnionGraphPattern then
    A := trnslGroupOrUnionGP(G)
else if G is a GraphGraphPattern then
    A := trnslGraphGP(G)
else if G is a GroupGraphPattern then
    A := trnslGroupGP(G)
return A
```

Transformation of GroupOrUnionGraphPattern

Algorithm 2 trnslGroupOrUnionGP(*G*)

Input: a GroupOrUnionGraphPattern *G*
with elements $e_1,...,e_n$

Output: a SPARQL algebra expression *A*

for $i = 1,\ldots,n$ **do**
    if *A* is undefined **then**
    A := translate(e_i)
else
    A := Union(A, translate(e_i))
return A

Transformation of `GraphGraphPattern`

Algorithm 3 trnslGraphGP(*G*)

Input: a `GraphGraphPattern` *G*
Output: a SPARQL algebra expression *A*

1: **if** *G* has the form `GRAPH IRI` *GroupGraphPattern* **then**
2: *A* := Graph(`IRI`, translate(*GroupGraphPattern*))
3: **else if** *G* has the form `GRAPH` *Var* *GroupGraphPattern* **then**
4: *A* := Graph(*Var*, translate(*GroupGraphPattern*))
5: **return** A

Transformation of GroupGraphPattern

Algorithm 4 trnslGroupGP(G)

Input: a GroupGraphPattern $G = (e_1,\dots,e_n)$

Output: a SPARQL algebra expression A

$A := Z$ // the empty pattern
$F := \varnothing$ // the empty filter
for $i = 1,\dots,n$ **do**
    if e_i is of the form FILTER(f) **then**
        $F := F \cup \{f\}$
    else if e_i is of the form OPTIONAL { P } **then**
        if translate(P) is of the form Filter(F',A') **then**
            $A :=$ LeftJoin(A, A' , F')
        else
            $A :=$ LeftJoin(A, translate(P), true)

Transformation of GroupGraphPattern (cntd)

else

A := Join(A, translate(e_i))

if F $\neq \emptyset$ **then**

A := Filter($\bigwedge_{f \in F} f$, A)

return A

Simplification of Algebra Expressions

- Groups with just one pattern (without filters) result in Join(*Z*, *A*) and can be substituted by *A*
- The empty pattern is the identity for joins:
  - Replace Join(*Z*, *A*) by *A*
  - Replace Join(*A*, *Z*) by *A*

Operators for Representing the Modifiers

Operator	Description
ToList(*M*)	Constructs from a multiset a sequence with the same elements and multiplicity (arbitrary order, duplicates not necessarily adjacent)
OrderBy(*M*, comparators)	sorts the solutions
Distinct(*M*)	removes the duplicates
Reduced(*M*)	may remove duplicates
Slice(*M*, *o*, *l*)	cuts the solutions to a list of length *l* starting from position *o*
Project(*M*, *vars*)	projects out the mentioned variables

Transformation of the Modifiers

Let q be a SPARQL query with pattern P and corresponding algebra expression A_P.
We construct an algebra expression A_q for q as follows:

① A_q := ToList(A_P)

② A_q := OrderBy(A_q, ($c_1, \ldots, c_n$)) if
q contains an ORDER BY clause
with comparators $c_1, \ldots, c_n$

③ A_q := Project(A_q, *vars*) if
the result format is SELECT with
vars the selected variables (* all variables in scope)

Transformation of the Modifiers

④ A_q := Distinct(A_q) if
the result format is SELECT and
q contains DISTINCT

⑤ A_q := Reduced(A_q) if
the result format is SELECT and
q contains REDUCED

⑥ A_q := Slice(A_q, *start*, *length*) if
the query contains OFFSET *start* or LIMIT *length*
where *start* defaults to 0 and
length defaults to $(|\, [[A_q]]_G \,| - start)$

Evaluation of the Modifiers

The algebra expressions for the modifiers
are recursively evaluated

- Evaluate the algebra expression of the operator
- Apply the operations for the solution modifiers to the obtained solutions