A relational algebra for SPARQL

Richard Cyganiak
Digital Media Systems Laboratory
HP Laboratories Bristol
HPL-2005-170
September 28, 2005*

RDF, semantic web, databases, query

The SPARQL query language for RDF provides Semantic Web developers with a powerful tool to extract information from large datasets. This report describes a transformation from SPARQL into the relational algebra, an abstract intermediate language for the expression and analysis of queries. This makes existing work on query planning and optimization available to SPARQL implementors. A further translation into SQL is outlined, and mismatches between SPARQL semantics and the relational approach are discussed.

* Internal Accession Date Only
Approved for External Publication

A relational algebra for SPARQL

Richard Cyganiak
richard@cyganiak.de

HP Labs, Bristol, UK

Abstract. The SPARQL query language for RDF provides Semantic Web developers with a powerful tool to extract information from large datasets. This report describes a transformation from SPARQL into the relational algebra, an abstract intermediate language for the expression and analysis of queries. This makes existing work on query planning and optimization available to SPARQL implementors. A further translation into SQL is outlined, and mismatches between SPARQL semantics and the relational approach are discussed.

The SPARQL query language is one of the latest additions to the Semantic Web toolbox. It provides powerful facilities to extract information out of large sets of RDF data. This report describes a transformation from SPARQL queries into the relational algebra, an intermediate language for the expression and analysis of queries that is widely used in the database area [2].

An RDF graph can be represented as a "table" with three columns, ?*subject*, ?*predicate* and ?*object*. Each row corresponds to one triple. Similarly, the result of a SPARQL SELECT query is a table of RDF nodes. These tables are *RDF relations*. The algebra described in this report consists of operators on RDF relations. Some of them are well-known from regular relational algebra, others are slightly modified to reproduce SPARQL semantics.

Figure 1 shows a SPARQL query and its translation into a relational operator tree.

Expressing SPARQL queries in relational algebra has several benefits:

- It makes a large body of work on query planning and optimization available to SPARQL implementers.
- It simplifies the problem of supporting SPARQL in database-backed RDF stores by splitting off considerations specific to the store's database layout. Section 3 outlines a translation into SQL.
- While this report assumes a simple relation of triples without further substructure as the basic building block of expressions, it might just as well be a "view" on a more complex source of triples, like, for example, a distributed union of RDF graphs, or an application-specific database mapped into RDF. Substituting their definitions into the operator tree, and subsequent use of expression transformation and query planning could yield efficient execution strategies for those complex sources of RDF data.

```
SELECT ?name ?email
WHERE {
   ?person rdf:type foaf:Person .
   ?person foaf:name ?name .
   OPTIONAL { ?person foaf:mbox ?email }
}
```

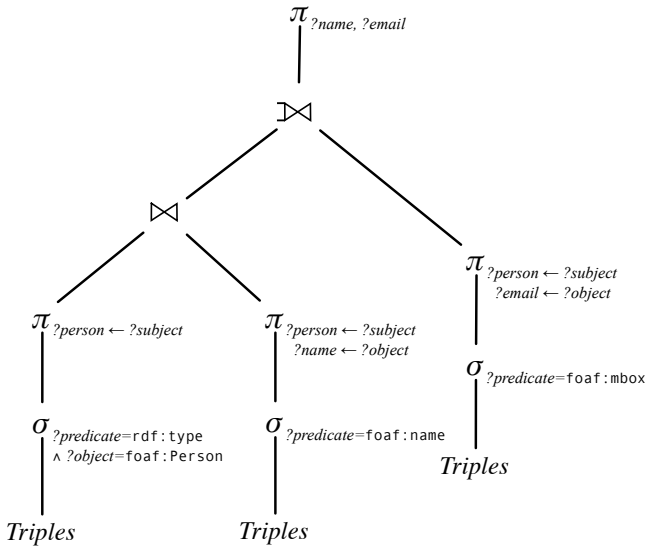


Fig. 1. A SPARQL query and its translation into a relational operator tree.

The framework presented here does not cover all of SPARQL. Only queries against the default graph are supported; an extension with full RDF dataset support is outlined in section 2.4, but not fully worked out. A translation of `FILTER` expressions into SQL remains as future work. Some corner cases are not easily translated; they are discussed in section 4.

1 A relational model over RDF terms

An **RDF term** is an IRI or an RDF Literal or a blank node, as defined in the RDF Concepts and Abstract Syntax document [4]. RDF terms are the scalars of the relational model presented here.

A **variable** is a query variable, as defined in the SPARQL document [5]. For the purposes of this report, variables are just names and are represented like this: ?*variable*.

An **RDF tuple** is a partial function from variables to RDF terms.

$$t = \begin{cases} ?person \rightarrow \texttt{<http://example.org/people\#Bob>} \\ ?name \rightarrow \texttt{"Bob"} \\ ?email \rightarrow \texttt{<mailto:bob@example.org>} \\ ?age \rightarrow \texttt{"42"} \end{cases}$$

An RDF tuple is quite different from an RDF *triple*. A triple always has three components with fixed names, *subject*, *predicate* and *object*, while an RDF triple can have any number of components with arbitrary names. A triple is a *statement* – it implies a semantic relationship between its components. A tuple doesn't carry meaning, it is just a container that maps some variables to some RDF terms.

Note, however, that any triple could be represented as a tuple:

$$t = \begin{cases} ?subject \rightarrow \texttt{<http://example.org/people\#Bob>} \\ ?predicate \rightarrow \texttt{ex:name} \\ ?object \rightarrow \texttt{"Bob"} \end{cases}$$

I will call this an **s/p/o tuple** because it has three components named *subject*, *predicate*, *object*. There is a one-to-one mapping from RDF triples to s/p/o tuples.

RDF tuples are just another term for SPARQL **solutions**. The term *tuple* is used universally in relational algebra and I will go with it in this document.

The variables in a tuple are called its **attributes**. An attribute is said to be **bound** if it is in the tuple's domain, and **unbound** otherwise. In the first example of this section, $?person$ is bound, and $?dateOfBirth$ is unbound because it is not in the tuple.

1.1 RDF relations

An **RDF relation** is a set of RDF tuples. It can be represented as a table. Each row is an RDF tuple and each column is an attribute, named by a variable.

?person	*?name*	*?email*	*?age*
`ex:Alice`	`"Alice"`		`"33"`
`ex:Bob`	`"Bob"`	`<mailto:bob@example.org>`	`"42"`

Not all tuples of a relation need to have the same attributes. An attribute can be unbound in some tuples. In the example, $?email$ is unbound in one tuple. This is similar to the special value NULL found in some definitions of relational algebra and SQL, but not quite the same, as discussed in section 4.

An RDF relation's **heading** is the set of variables that may be bound in any of its tuples. Intuitively speaking, the heading contains all column names. The heading can usually obtained by looking at the attributes present in each tuple.

Defining an explicit heading that is independent from the tuples allows reasoning about operations on relations without looking at the data they contain.

In the full relational model, every attribute has its own **domain**. In the presented model, the domain of every attribute is simply the set of all RDF terms. However, an extension that takes possible values of individual columns into account would have benefits. For example, an implementation might choose a more efficient strategy for handling an attribute whose domain is the set of all RDF literals with datatype `xsd:integer`.

A **graph relation** is any relation whose heading is {?*subject*, ?*predicate*, ?*object*}. Each of the attributes must be bound in every tuple, ?*subject* must always be a literal, and ?*predicate* must always be an IRI. In other words, every tuple must be an s/p/o tuple.

?*subject*	?*predicate*	?*object*
`ex:Alice`	`ex:name`	`"Alice"`
`ex:Alice`	`ex:age`	`"33"`
`ex:Bob`	`ex:name`	`"Bob"`
`ex:Bob`	`ex:mbox`	`<mailto:bob@example.org>`
`ex:Bob`	`ex:age`	`"42"`

As stated above, every s/p/o tuple has a corresponding RDF triple. It follows that every graph relation has an equivalent RDF graph, and vice versa. The graph relation above corresponds to this RDF graph:

```
ex:Alice ex:name "Alice" .
ex:Alice ex:age "33" .
ex:Bob ex:name "Bob" .
ex:Bob ex:mbox <mailto:bob@example.org> .
ex:Bob ex:age "42" .
```

The rest of the section defines operators on RDF relations.

1.2 Selection

Selection (σ), sometimes also called *restriction*, is an unary operator that selects only those tuples of a relation for which a propositional formula holds. The propositions are assumed to have the expressivity of SPARQL `FILTER` expressions [5].

$$\sigma_{?age \geq 18 \vee \texttt{bound}(?parent)}(R)$$

A simple form of Selection is the **constant attribute selection**, which accepts all tuples with a constant value on some attribute. The example uses this operator to filter a graph relation:

$$\sigma_{?predicate = \texttt{ex:name}}(R)$$

The heading of any $\sigma_p(R)$ is the same as the heading of R.

1.3 Projection and Rename

The **projection** operator (π) restricts a relation to a subset of its attributes. The **rename** operator (ρ) renames an attribute. For the purposes of this report, I will combine them into a **projection/rename** operator. This is a shorthand notation to keep the operator trees smaller.

$$\pi_{\substack{?person \leftarrow ?subject \\ ?name \leftarrow ?object}}(R)$$

This projection contains only the `?subject` and `?object` attributes of relation R, renamed to `?person` and `?name`.

1.4 Inner Join and Left Outer Join

The **inner join** ($\bowtie$) joins two relations on their shared attributes. $A \bowtie B$ contains all combinations of a tuple from A and a tuple from B, minus those where the shared attributes are not equal.

A shared attribute is any attribute that is bound in both *tuples*. This is different from regular relational algebra, where a shared attribute would be any that occurs in both *relations*. This is discussed in more detail in section 4.2.

The join's heading is the set union of the headings of A and B.

A

?person	*?name*
`ex:Alice`	`"Alice"`
`ex:Bob`	`"Bob"`

B

?person	*?parent*
`ex:Bob`	`ex:Charles`
`ex:Bob`	`ex:Dorothy`

$A \bowtie B$

?person	*?name*	*?parent*
`ex:Bob`	`"Bob"`	`ex:Charles`
`ex:Bob`	`"Bob"`	`ex:Dorothy`

The `ex:Alice` from the first relation row does not appear in the join because there is no row in the second relation with the same value for *?person*.

The result of a **left outer join** ($\text{:}\bowtie$)[1] additionally contains all those tuples from the first relation that have no matching tuple in the second:

$A \text{:}\bowtie B$

?person	*?name*	*?parent*
`ex:Alice`	`"Alice"`	
`ex:Bob`	`"Bob"`	`ex:Charles`
`ex:Bob`	`"Bob"`	`ex:Dorothy`

[1] I was unable to produce the correct symbol. The "dots" are supposed to be connected to the "bowtie".

I introduce two additional operators, the **conditional inner join** and the **conditional left outer join**. They drop the combined tuples that do not match a propositional formula. The difference between $A \bowtie_p B$ and $A \bowtie (\sigma_p(B))$ is that the proposition p can use attributes from both A and B. The operators are used to overcome the "`FILTER` scope problem" discussed in section 4.4.

1.5 Union

The **union** ($\cup$) of two relations A and B is the set union of the tuples of A and B. Unlike in regular relational algebra, the headings of A and B do not need to be identical. Attributes unbound in the tuples of one relation are simply left unbound in the result relation. This is sometimes called an *outer union*.

The heading of $A \cup B$ is the set union of the headings of A and B.

$A \cup B$

?person	*?name*	*?parent*
`ex:Alice`	`"Alice"`	
`ex:Bob`	`"Bob"`	
`ex:Bob`		`ex:Charles`
`ex:Bob`		`ex:Dorothy`

1.6 Set Difference

This operator is not defined here because it is not necessary for the purposes of this report and can not generally be expressed in SPARQL. I note that it would be required for relational completeness.

2 A relational definition of SPARQL

The SPARQL specification formally defines the semantics of the query language in terms of *solutions*. This is just a different term for an RDF tuple – a partial function from variables to RDF terms. The formal definitions describe the properties that a tuple must have to be a solution to a pattern when matched against a particular RDF dataset. This style of definition makes it easy to check if a given tuple is a solution or not, but doesn't tell how to find all solutions.

This section provides an alternative definition that produces the same results (although no proof is offered), but better informs implementation. This is somewhat analogous to the relational calculus and relational algebra in the relational model. Both are equivalent, but the calculus describes the properties of solutions, while the algebra builds the result from the data through a set of operators that can be implemented reasonably straightforward.

The main part of a SPARQL query is the *query pattern*. It is usually built out of several *graph patterns*, which can be combined recursively to form arbitrarily complex query patterns. This section provides translations for the different kinds

of graph patterns into relational algebra expressions. No translation for the *basic graph pattern* is given because it is just a *group* of *triple patterns*. The result of translating any graph pattern into a relational expression is called its *pattern relation*.

In the SPARQL document's language, matching the entire query pattern against a graph produces a *solution sequence*. This is the same as the pattern relation of the entire query pattern.

All the remaining bits of SPARQL processing, like `DISTINCT`, `ORDER BY`, and the `CONSTRUCT`, `DESCRIBE` and `ASK` result forms, can then be applied to the solution sequence, as defined in the SPARQL specification.

2.1 Triple patterns

The pattern relation of a triple pattern is obtained from the graph relation by constant attribute selection on the concrete nodes and by projection/renaming on the variable nodes. The SPARQL triple pattern

```
{ ?person foaf:name ?name }
```

translates to this relational operation:

$$\pi_{\substack{?person \leftarrow ?subject \\ ?name \leftarrow ?object}} (\sigma_{?predicate = \texttt{foaf:name}}(Triples))$$

2.2 Group patterns, `OPTIONAL` and `FILTER`

SPARQL group patterns are used to build complex patterns out of simple parts. The order of the subpatterns within the group does not matter, with the exception of `OPTIONAL`s, where the order is significant because a match of one `OPTIONAL` can cause a later one that would also have matched to fail.[2] The parts of a group pattern are translated in this order:

1. Inner join all triple patterns, `UNION`s and `GRAPH`s. Their relative order does not matter because $A \bowtie B = B \bowtie A$ and $A \bowtie (B \bowtie C) = (A \bowtie B) \bowtie C$.
2. Then, left outer join all `OPTIONAL` subpatterns to the result of 1, in the order they occur in the query, with the optional subpattern becoming the right side of the outer join. `OPTIONAL`s must be handled with lower precedence than the required subpatterns because otherwise, a matching optional part could cause a required part to fail.
3. Finally, take each `FILTER`'s condition and apply them as selection operators to the result of 2. They must be applied last because the condition can use variables from anywhere in the group. If there are multiple `FILTER`s in the group, their relative order does not matter because $\sigma_{p1}(\sigma_{p2}(R)) = \sigma_{p2}(\sigma_{p1}(R))$.

[2] The SPARQL specification is not entirely clear on this subject, but I believe that this is the intended behaviour of group patterns.

The group

```
{ p1 . FILTER (p) . OPTIONAL { p2 } . p3 . }
```

translates to:

$$\sigma_p((p1 \bowtie p3) :\bowtie p2)$$

$p1$, $p2$ and $p3$ are arbitrary sub-patterns in the SPARQL expressions, and their pattern relations in the operator tree.

This translation does not work for some cases discussed in section 4.

2.3 `UNION` patterns

A `UNION` pattern matches if any of its subpatterns matches. Its pattern relation is obtained by connecting the relations of the subpatterns in any order with the union operator.

```
{ p1 } UNION { p2 }
```

translates to:

$$p1 \cup p2$$

2.4 `GRAPH` and the RDF dataset

This report generally assumes that the SPARQL query is evaluated against an individual RDF graph – the default graph of an RDF dataset. This subsection outlines an extension that works with full RDF datasets.

All triple patterns that are inside a `GRAPH` pattern are extended to `quad patterns`. The fourth element is the graph name provided by the `GRAPH` construct. It may be a variable or an IRIref.

Instead of the graph relation, we would have a `dataset relation` with a fourth attribute, ?*graph*, which is bound to the graph name for triples in named graphs, and bound to some special value `DEFAULTGRAPH` for triples in the default graph. Triple patterns match only when ?*graph* equals `DEFAULTGRAPH`, quad patterns only if the fourth element matches ?*graph*.

The pattern

```
GRAPH ?g { p1 }
```

would translate to:

$$\pi_{?g \leftarrow ?graph}(Dataset) \bowtie p1$$

This accounts for patterns like this:

```
GRAPH ?g { OPTIONAL { ?a :b :c } }
```

Somewhat unintuitively, this pattern binds $?g$ to all graph names in the dataset even if the optional part never matches.

Blank nodes in an RDF dataset can never be shared between graphs. If labels are used to identify blank nodes, then relabelling might be necessary when graphs are added to the dataset.

3 Mapping to SQL

This section outlines a translation from the relational algebra into SQL statements. Some parts will be left a bit vague to keep the translation independent from the database layout used to represent RDF triples and nodes. Also, the focus is not to produce optimized SQL, but rather to keep the translation straightforward.

The recursive nature of SPARQL's graph patterns requires some form of nesting in the generated SQL. An implementation may (and probably should) use a mix of different SQL language features to construct nested queries:

Temporary tables: Intermediate results can be stored in temporary tables. This approach is the most flexible. The temporary tables can be processed with non-database code[3] or used multiple times in different parts of the bigger query. It also produces quite manageable SQL statements because one big SPARQL query translates into several small SQL queries. In general the result of processing any subtree can be stored in a temporary table. This approach is used in 3Store [3].

Nested `SELECT`s: This is the approach used here. Bracketed `SELECT` statements are used in the `FROM` clause of SQL queries. The translation is quite straightforward since the resulting queries closely mirror the structure of the relational algebra expression. Unfortunately, they are also very unwieldy and unlikely to be executed with good performance. At least some "flattening" is advisable.

Bracketed `JOIN`s: Several aliases of the triple table are connected by `JOIN`s and `LEFT JOIN`s. Brackets around the joins ensure the right order. This approach produces concise SQL statements, and informal testing with MySQL indicates that it produces the best performance. An algorithm for translating into this form is more complex because column names from the base tables are carried all the way up through the levels of nesting, and `WHERE` conditions of lower levels must be put into the parent's `JOIN` condition. The union operator can not be translated with this approach. Bracketed `JOIN`s are not supported in MySQL prior to version 5.

The rest of the section provides translations for the relational operators.

[3] This allows support for expressions that cannot be evaluated inside the database, like extension functions or certain forms of regular expressions.

3.1 Projection and Rename

This operator translates straightforwardly to a `SELECT` statement with column aliases. The expression

$$\pi_{\substack{?person \leftarrow ?subject \\ ?name \leftarrow ?object}}(R)$$

translates to this:

```
SELECT R.subject AS person, R.object AS name FROM (...) AS R;
```

Into the brackets goes the translation of R to SQL.

3.2 Selection

The translation of SPARQL's expression syntax to SQL expressions depends strongly on the way nodes are encoded in the database and the SQL types of the database columns. I don't attempt a full translation and only give some general notes. Harris [3] has some additional material.

The translated expression will be used in the `WHERE` clause of a `SELECT` statement.

$$\sigma_{?price<30}(R)$$

```
SELECT * FROM (...) AS R WHERE R.price<30;
```

If R translates to a `SELECT` statement, then the `WHERE` condition can be added directly to that `SELECT`, avoiding one level of nesting. If R translates to a `UNION` statement, then the condition can be applied to all of its branches.

3.3 Inner Join

The inner join of RDF relational algebra is roughly equivalent to an SQL inner join over the variables shared by both relations. The translation of this expression:

$$\pi_{?a,?x}(R_1) \bowtie \pi_{?a,?y}(R_2)$$

to SQL looks like this:

```
    SELECT R1.a, R1.x, R2.y
      FROM (...) AS R1
NATURAL JOIN (...) AS R2;
```

SQL's `NATURAL JOIN` joins on the columns that are present on both sides. The translations of R_1 and R_2 to SQL go into the brackets.

For reasons discussed in the next section, this simple translation works only if the shared variable is always bound on both sides. This is the case for most, but not all SPARQL inner joins. If the variable might be unbound on either side, then a more complex translation is required.

A NULL in an SQL join causes the row to be rejected, but an unbound variable in a SPARQL join does not. A SPARQL join rejects only rows where the variable is bound to different values on both sides. The following translations reproduce this behaviour in SQL.

If ?a may be unbound on the left:

```
SELECT R2.a, R1.x, R2.y
  FROM (...) AS R1
  JOIN (...) AS R2
    ON (R1.a=R2.a OR R1.a IS NULL);
```

If ?a may be unbound on the right:

```
SELECT R1.a, R1.x, R2.y
  FROM (...) AS R1
  JOIN (...) AS R2
    ON (R1.a=R2.a OR R2.a IS NULL);
```

If ?a may be unbound on both sides, then SQL's `COALESCE` function is used to gather the value from the non-NULL side:

```
SELECT COALESCE(R1.a, R2.a) AS a, R1.x, R2.y
  FROM (...) AS R1
  JOIN (...) AS R2
    ON (R1.a=R2.a OR R1.a IS NULL OR R2.a IS NULL);
```

Implementations can simply always use the last translation. It is identical to the first one if `R1.a` and `R2.a` are never NULL. But this does not perform so well.

Instead, implementations might choose track the attributes that may be unbound in each relation throughout the operator tree, and then choose the appropriate translation. The attributes of the graph relation are always bound. The operators propagate this property as follows:

Inner join: An attribute may be unbound if it is unbound on both sides of the join; otherwise, it is always bound.

Left outer join: An attribute is always bound if it is always bound on the left side; all other attributes may be unbound.

Union: An attribute is always bound if it is always bound in both branches of the union; otherwise, it may be unbound.

All other operators: The bound/unbound status is preserved.

The preceding examples have assumed that only one join, `?a`, is shared between the relations. The next example joins on variables `?a`, `?b` and `?c`, where `?b` might be unbound on the left-hand side, and `?c` might be unbound on both sides.

$$\pi_{?a,?b,?c,?x}(R_1) \bowtie \pi_{?a,?b,?c,?y}(R_2)$$

(The expression doesn't capture the fact that some variables may be unbound.)

```
SELECT R1.a, R2.b, COALESCE(R1.c, R2.c) AS c, R1.x, R2.y
  FROM (...) AS R1
  JOIN (...) AS R2
    ON R1.a=R2.a
   AND (R1.b=R2.b OR R1.b IS NULL)
   AND (R1.c=R2.c OR R1.c IS NULL OR R2.c IS NULL);
```

Finally, to translate a conditional inner join, its condition is added to the `ON` clause with the `AND` operator.

3.4 Left Outer Join

The translation is analogous to the inner join.

$$\pi_{?a,?x}(R_1) \mathbin{:\bowtie} \pi_{?a,?y}(R_2)$$

If $?a$ is always bound on both sides:

```
          SELECT R1.a, R1.x, R2.y
            FROM (...) AS R1
NATURAL LEFT JOIN (...) AS R2;
```

If $?a$ may be unbound on the left:

```
   SELECT COALESCE(R1.a, R2.a) AS a, R1.x, R2.y
     FROM (...) AS R1
LEFT JOIN (...) AS R2
       ON (R1.a=R2.a OR R1.a IS NULL);
```

If $?a$ may be unbound on the right:

```
   SELECT r1.a, r1.x, r2.y
     FROM (...) AS r1
LEFT JOIN (...) AS r2
       ON (r1.a=r2.a OR r2.a IS NULL);
```

If $?a$ may be unbound on both sides:

```
   SELECT COALESCE(R1.a, R2.a) AS a, R1.x, R2.y
     FROM (...) AS R1
LEFT JOIN (...) AS R2
       ON (R1.a=R2.a OR R1.a IS NULL OR R2.a IS NULL);
```

To translate a conditional outer join, its condition is added to the `ON` clause with the `AND` operator.

3.5 Union

SQL's `UNION` can be used. Attributes used only on one side of the union have to be filled with NULLs and all attributes must be brought into the same order.

$$\pi_{?a,?b}(R_1) \cup \pi_{?b,?c}(R_2)$$

```
SELECT R1.a, R1.b, NULL AS c FROM (...) AS R1
 UNION
SELECT NULL AS a, R2.b, R2.c FROM (...) AS R2;
```

The performance of joins on `UNION` results appears to be poor on many database engines. This makes unions prime candidates for optimization by transforming the expression tree: Inner joins can be distributed over both sides of the union, turning $A \bowtie (B \cup C)$ into $(A \bowtie B) \cup (A \bowtie C)$.

3.6 Flattening nested `SELECT`s

`JOIN` and `LEFT JOIN` expressions involving nested `SELECT`s can be flattened into simpler SQL expressions.

This expression, which is already somewhat simplified from the operator tree in figure 2:

```
   SELECT sub1.name, sub2.email
     FROM
  (SELECT t1.subject AS person,
          t1.object AS name
     FROM Triples AS t1
    WHERE t1.predicate='foaf:name')
       AS sub1
LEFT JOIN
  (SELECT t2.subject AS person,
          t2.object AS email
     FROM Triples AS t2
    WHERE t2.predicate='foaf:mbox')
       AS sub2
       ON sub1.person=sub2.person
```

can be flattened into this:

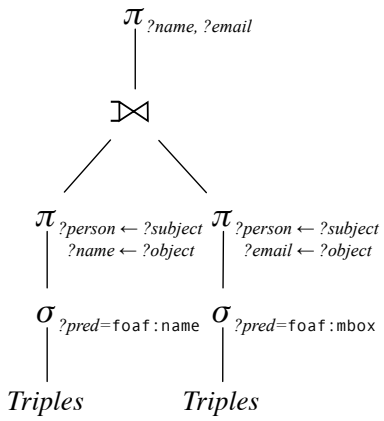


Fig. 2. This operator tree can be translated into a "flat" SQL query.

```
   SELECT t1.object AS name, t2.object AS email
     FROM Triples AS t1
LEFT JOIN (Triples AS t2)
       ON t1.subject=t2.subject
      AND t1.predicate='foaf:name'
      AND t2.predicate='foaf:mbox'
```

References to variables from a sub-expression (`sub1.person`, `sub1.name`, `sub2.email`) are substituted by their definitions (`t1.subject`, `t1.object`, `t2.object`).

The `WHERE` clause of the inner `SELECT`s become part of the join condition. This is important for `LEFT JOIN`s because the condition should apply only to rows where the right-hand side matches, not to all rows.

Note the brackets around `Statements AS t2` in the flattened expression. If this clause was itself a nested sub-expression, then the brackets would determine the execution order of the `JOIN`s. Inner joins are commutative and can be evaluated in any order, but left outer joins must be evaluated in the order implied by the operator tree.

4 Mismatches between the relational model and SPARQL semantics

Most SPARQL queries can be translated into a relational form without much trouble, but some issues – semantic mismatches and interesting corner cases – deserve closer attention.

Despite being in the Last Call stage of the W3C recommendation track, the SPARQL query language document currently lacks mathematical rigor and fails

to accurately define the semantics for some cases. I expect this to be addressed before SPARQL becomes a W3C recommendation. This report may need revision in response to these changes.

4.1 Unbound variables, the NULL value and headings

SPARQL and the relational algebra take different approaches to the problem of missing information.

The common approach in relational algebra is to indicate an unknown value using the special value NULL. Also, relations have an explicit attribute list in the heading, and every tuple must have exactly these attributes, either with a value or with NULL.

SPARQL doesn't use a special value to indicate missing information, but simply leaves variables unbound. There's no explicit heading. The SPARQL model does not, for example, distinguish between an `OPTIONAL` variable that is unbound in some solutions, and a variable that is not used in the query at all.

The model presented attempts to be compatible with both accounts. There is no special NULL value, but relations have an explicit heading. An attribute that is not in the heading must not be bound in any tuple, but an attribute that is in the heading may be unbound in some tuples.

4.2 Join behaviour with missing information

To accommodate SPARQLs handling of missing information, the join operators have to be defined slightly different from their counterparts in regular relational algebra. In a regular join, a NULL in any join attribute causes a tuple combination to be rejected.

In RDF relational algebra, the combination is rejected only if there is a *conflict* – an attribute is bound on both sides, but to different values. If a shared attribute is unbound on one side, then the value from the other side is taken for the result tuple. If a shared attribute is unbound on both sides, then the attribute is unbound in the result tuple.

This accounts for cases like the following:

```
SELECT ?resource ?titleOrLabel
WHERE {
   ?resource a ex:MyClass .
   OPTIONAL { ?resource dc:title ?titleOrLabel }
   OPTIONAL { ?resource rdfs:label ?titleOrLabel }
}
```

This query returns all instances of `ex:MyClass`, and their title if they have one. The title can come either from the `dc:title` or `rdfs:label` property, with preference for `dc:title` because it is first. The corresponding operator tree is shown in figure 3. The first `OPTIONAL` corresponds to the lower join operator, the second `OPTIONAL` corresponds to the upper one.

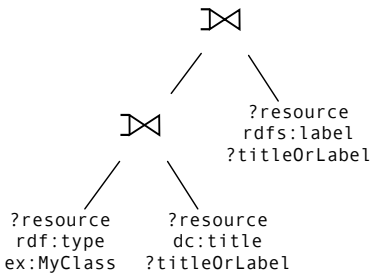
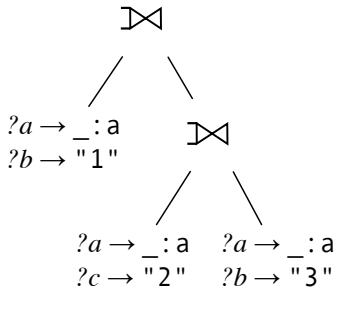


Fig. 3. Multiple `OPTIONAL`s introduce the same variable.

The "unbound variable causes join failure" semantics would only match resources that have the same `dc:title` and `rdfs:label`. The second `OPTIONAL` would never have any effect.

The "only conflicts cause join failure" semantics produces the desired behaviour. If a resource has one of `dc:title` and `rdfs:label`, then the join on *?titleOrLabel* takes whatever side is bound. If a resource has both, and they are different, then the join fails, but the `dc:title` side is taken anyway because it is a left outer join.

4.3 The "Nested `OPTIONAL`s" Problem




```
[] :p1 "1"; :p2 "2"; :p3 "3".

SELECT ?b ?c
WHERE {
   ?a :p1 ?b
   OPTIONAL {
      ?a :p2 ?c .
      OPTIONAL { ?a :p3 ?b }
   }
}
```

Fig. 4. The "Nested `OPTIONAL`s" problem: Which of the left joins will fail?

Unfortunately, the join rule stated above does not fully reproduce SPARQL semantics. The problem, illustrated in figure 4, occurs in a corner case where two `OPTIONAL`s are nested, and a variable is used both outside the `OPTIONAL`s and inside the inner one, but not in the outer one.

The question is: If ?*b* can only be bound to two different values at its two occurrences, which of the left joins will fail? The question is important because ?*c* will be bound to 2 if the lower one fails, and unbound if the upper one fails.

According to the relational model presented here, the upper one will fail. When evaluating the lower join, there's no conflict and ?*b* will be bound to 3. At the upper join, there's a conflict between the two different values bound to ?*b*, and thus the join will fail and the result will be the tuple from the left relation (figure 5).

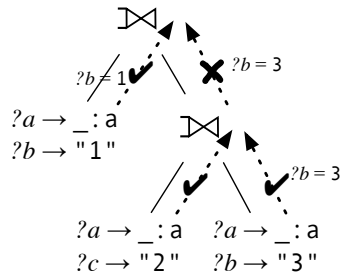


Fig. 5. Nested `OPTIONAL`s with relational processing: Bindings travel "up", the upper join fails.

According to the current SPARQL semantics, the lower one will fail. The inner `OPTIONAL` will fail because ?*b* has "already been bound earlier" to 1, and cannot be bound again to something different. The outer `OPTIONAL` doesn't cause a conflict then and will bind ?*c* (figure 6).

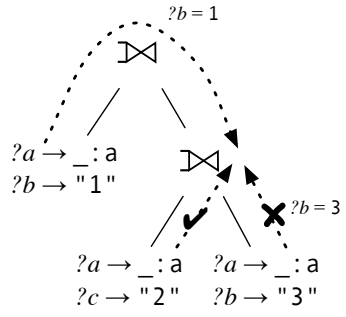


Fig. 6. Nested `OPTIONAL`s with SPARQL processing: Bindings travel "left to right", even down the tree.

Essentially, looking at the operator tree, solutions are assembled left-to-right in the SPARQL model and bottom-to-top in the relational model. This mismatch is not easily reconciled. Fortunately, this is a rather obscure case that is not likely to occur often in practice.

4.4 The "FILTER scope" problem

In SPARQL, `FILTER` expressions are free to use any variable, regardless of where the `FILTER` is located. They can use variables from outside of the current group pattern. This is a problem for the relational approach because a selection in the operator tree has no access to variables from outside of its subtree.

There are cases where access to variables from other parts of the tree is beneficial:

```
SELECT ?x ?type ?name
WHERE { ?x rdf:type ?type .
        OPTIONAL { ?x foaf:name ?name .
                   FILTER (?type == foaf:Person) }
}
```

This query returns all resources, their types, and the name only if it is a person. The operator tree is shown on the left in figure 7. ?*type* is only known in the left subtree, and the selection on ?*type* in the right subtree does not work.

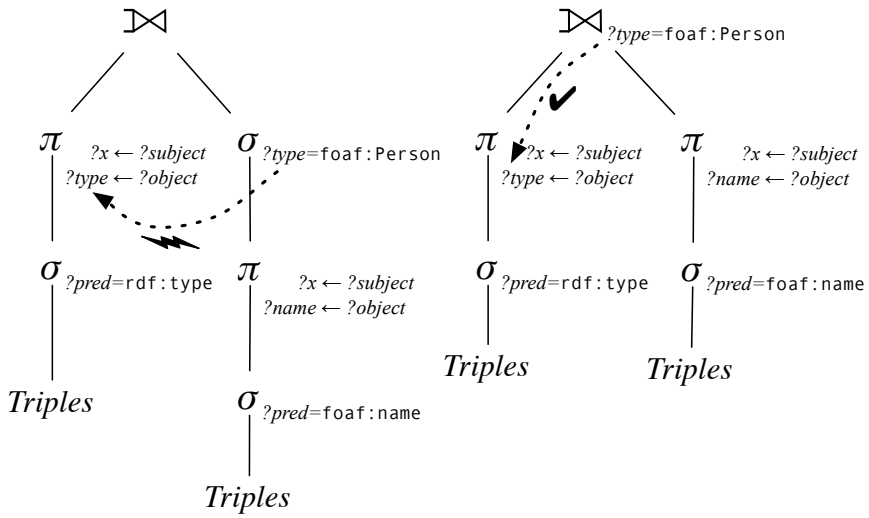


Fig. 7. The `FILTER` scope problem. ?*type* is bound in the left subtree; its value is not available to the selection. The right side shows a solution: The selection can be pulled up to form a conditional left join.

The specific case in figure 7 – an expression inside an optional group refers to a variable bound outside – might be the most common incarnation of this problem. A special translation from SPARQL to relational algebra can account for this case: The selection is pulled up into the next join to form a conditional join. This works if the out-of-scope variable is defined in the other branch of that join.

The question if the other cases, where a variable's value is accessible only even further up the operator tree, can also be solved by re-arranging the operator tree, remains as future work.[4]

5 Acknowledgements

This report owes much to the work of Benjamin Nowack, Steve Harris and Jörg Gabers. I couldn't have worked on this topic without Chris Bizer and his D2RQ [1]. Andy Seaborne's comments, support, and patience were invaluable.

References

1. C. Bizer, R. Cyganiak, and J. Gabers. *D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs.* Freie Universitt Berlin, http://www.wiwiss.fu-berlin.de/suhl/bizer/d2rq/.
2. C. J. Date. *An Introduction to Database Systems.* Addison Wesley, eighth edition, 2003.
3. S. Harris. Sparql query processing with conventional relational database systems, 2005. Submitted to *International Workshop on Scalable Semantic Web Knowledge Base System (SSWS 2005).* http://eprints.ecs.soton.ac.uk/11126/.
4. G. Klyne and J. J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax - W3C Recommendation*, 2004. http://www.w3.org/TR/rdf-concepts/.
5. E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF - W3C Working Draft 21 July 2005*, Jul 2005. http://www.w3.org/TR/2005/WD-rdf-sparql-query-20050721/.

[4] The `FILTER` scope problem does not occur if a `FILTER` variable is bound in another branch of a union because union branches can not affect each other.