Title: Converting SPARQL to SQL

Author: Fred Zemke
Date: October 12, 2006

References

[SPARQL CR] "SPARQL query language for RDF", Candidate Recommendation, http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/

[rq24] "SPARQL query language for RDF", Editor's draft, http://www.w3.org/2001/sw/DataAccess/rq23/24 (the version I printed was dated 2006/06/12)

[First attempt] Fred Zemke, "An attempt at a formal semantics for SPARQL"

[Constructive/destructive] Fred Zemke, "SPARQL semantics: constructive or destructive?"

[Constructive mapping semantics] Fred Zemke, "Constructive mapping semantics for SPARQL"

[SPARQL to trees] Fred Zemke, "Converting SPARQL syntax to trees"

1. Introduction

This paper continues the work begun in [SPARQL to trees] and [Constructive mapping semantics]. Those papers proposed a formal semantics for SPARQL in two phases:

1. [SPARQL to trees] shows how to convert SPARQL syntax to a tree representation, and
2. [Constructive mapping semantics] provides the semantics for the trees.

There are minor, resolvable differences between the tree representations in those two papers.

This paper shows how to convert a SPARQL query tree (generated according to the algorithm of [SPARQL to trees]) to an SQL statement, in effect making SQL the pseudocode or metalanguage for defining the semantics of SPARQL. I believe and intend that

[SPARQL to trees] + [Constructive mapping semantics]

is equivalent to

[SPARQL to trees] + this paper.

2. Preliminaries

2.1 Terminology

I have found the following terminology convenient:

bindable: a variable or blank node identifier

graph selector: the VarOrBlankNodeOrIRIref that is the first argument of a GraphGraphPattern (rule [25]). Elsewhere I have suggested that permitting a blank node identifier as a graph selector is inconsistent with using entailment as a general framework. Andy Seaborne has implemented this suggestion (http://lists.w3.org/Archives/Public/public-rdf-dawg/2006OctDec/0055.html item 5). However, the work in this paper is independent of that question.

mandatory pattern: the first operand of an OPTIONAL

supplementary pattern: the second operand of an OPTIONAL

constant pattern: a pattern having no bindables. A constant pattern has a single solution with no bindings, if the constant pattern matches; otherwise it has no solution. The empty pattern is a special case of a constant pattern. A use of nonempty constant patterns is shown by the query

```
WHERE { ?g GRAPH { :s :v :n } }
```

which asks which named graphs contain the triple.

2.2 Font conventions

The algorithm will construct a lot of SQL text. Constructed SQL text will be shown on indented monospace lines. SQL text that is shown without italics should be taken literally. SQL text that is italicized represents a point where text substitution occurs; the italicized variable is the name of previously constructed text. Frequently the substitution text is the result of a function; in that case, the function name is italicized but the function arguments typically are not.

2.3 Graphs

I assume that graphs are stored as relational tables with three columns, which will be called Subject, Verb and Object in this paper. I assume that (Subject, Verb, Object) constitutes the primary key (this insures that the graph is a mathematical set, with no duplicates, and no row/column intersection is null).

I also assume that there is a function that maps graph names (IRIs) to table names. This function also maps the default graph to a table name.

Alternatively, all known graphs can be stored in a single table, with a fourth column for the graph IRI. With this technique, the primary key needs to include the graph IRI column as well as the Subject, Verb and Object.

I assume that all three or four columns are stored as character strings, perhaps a large VARCHAR type, or even CLOB. Let *CharType* be the syntax for declaring this type in SQL.

2.4 Blank nodes in graphs

I assume that all blank nodes in graphs are represented by blank node identifiers (character strings) that are unique across all graphs in the query. In the event that the graphs in the query were not constructed according to that principle, it can be imposed after the fact in SQL using the following expression syntax:

```
CASE WHEN value LIKE '_:%'
     THEN '_:' || 'GraphId(IRI)' || '_'
          || SUBSTRING (value FROM 3)
     ELSE value END
```

where *GraphId* is a function that maps the graph's IRI name to a unique identifier containing no underscore.

This massages all blank node identifiers so that they begin with a unique identifier for the graph that contains them, while passing any IRI values through unchanged. Thus it becomes impossible for two blank nodes in separate graphs to be equal.

2.5 Default graph assignment

[Constructive mapping semantics] defines the notion of a path through a tree, and then constructions built on those paths. A construction consists of two functions, one which assigns a default graph to each activated node of a tree, and one which assigns a mapping (solution) to each activated node.

In this paper, I found it possible to isolate the default graph assignment from the path by making the default graph assignment a total function on the tree. In this way, default graph assignments can be constructed prior to constructing solutions (instead of integrating the construction of the default graph assignment with the construction of the solution, as in [Constructive mapping semantics]).

Formally, let D = { G, (<u_1, G_1>), . . . , (<u_g>, G_g) } be an RDF dataset as defined in [rq24] section 9 "RDF dataset". I assume that the default graph G has an IRI u_0, purely as a device to reference the default graph (this does not alter the fact that the IRI of the default graph is not visible in the query unless the default graph is also an explicitly named graph).

I define a default graph assignment as a function DGA from a tree Tree to the set { u_0, . . ., u_g } satisfying the following properties:

1. If Node is the root node of Tree, then DGA(Node) = u_0.
2. If Node is a non-root node of Tree, let Parent be the parent of Node.

   a) If Parent is a GRAPH node, then:

i) If the first child of Parent (the graph selector) is an IRI and Parent = u_i for some i, then DGA(Node) = u_i.

ii) If the first child of Parent is a bindable, then DGA(Node) = u_i for some i between 1 and g.

[NOTE to the proposal reader: Andy Seaborne's announced change http://lists.w3.org/Archives/Public/public-rdf-dawg/2006OctDec/0055.html item 5 implies that the the preceding rule can be simplified to "If the first child of Parent is a variable, ...".]

b) Otherwise, DGA(Node) = DGA(Parent)

The set of all default graph assignments of a tree is finite, and may be enumerated by walking down the tree from root to leaves. Whenever a GRAPH node with a non-constant graph selector is encountered, it is necessary to explode the number of default graph assignments by the number of named graphs. If there are no non-constant graph selectors, then there is only one default graph assignment.

To handle some corner cases, it may happen that

1. A SPARQL query contains a GraphGraphPattern with an IRI as the graph selector, but there is no named graph associated with that IRI.
2. A SPARQL query contains a GraphGraphPattern with a bindable as the graph selector, but there is no named graph at all.

In these cases, it is impossible to create a default graph assignment. I propose that these cases should be treated as syntax errors, since they cannot match. However, if it is desired to permit these cases, then the algorithm in this paper can be fixed by assuming an empty graph with some IRI that is distinct from all u_i. A query performed on an empty graph will have no matches.

Let d be the number of default graph assignment of Tree. Let DGA_1, . . ., DGA_d be those default graph assignments.

2.6 Overall structure of the generated SQL query

I found it convenient to create an SQL subquery for every default graph assignment and every interior node of the tree. The bulk of the algorithm describes how to construct such subqueries.

The generated SQL is admittedly verbose, and the reader will probably observe less verbose ways of expressing every example presented in the paper. The purpose in this paper is merely to present a completely general algorithm, not an optimal one.

The WITH clause of SQL may be used to assign an identifier to a subquery. Then the subquery can be referenced by its name in FROM clauses later in the overall query. For example

```
WITH T1 AS (SELECT C1, K1 FROM Table1 WHERE K1 > 20),
```

```
     T2 AS (SELECT Table2.C2 FROM Table2, T1
            WHERE T1.K1 = Table2.K2)
SELECT *
FROM T2
```

This example defines T1, then defines T2 using the definition of T1, and finally performs a query against T2.

The WITH clause simplifies the proposal in this paper. However, the WITH clause is not a mandatory part of the SQL standard. When using an SQL implementation that does not support the WITH clause, the same effect can be achieved by nesting subqueries in-line. The example above is equivalent to

```
SELECT *
FROM (SELECT Table2.C2
      FROM Table2, (SELECT C1, K1
                    FROM Table1
                    WHERE K1 > 20) AS T1
      WHERE T1.K1 = Table2.K2)
```

2.7 Node numbering

I assume some enumeration {$Node_1$, $Node_2$, . . ., $Node_n$} of the nodes of Tree, with the property that if $Node_i$ is a proper descendant of $Node_j$, then i > j. Thus the root is necessarily $Node_1$, and other nodes can be numbered by walking down the tree in some fashion. The subqueries to be generated will be arranged in *descending* node number order (placing the subquery for the root last) so that the logical precursors to each subquery will defined earlier in the WITH clause.

Default graph assignments are placed in an arbitrary order, since no default graph assignment depends on another.

2.8 Node names

Every pair (DGA_i, $Node_j$) is given a node name. The node name is used in two forms, as an SQL identifier and as a SQL character string literal:

1. The identifier form of the node name is used as the name of the SQL subquery that discovers the solutions of $Node_j$ using DGA_i as the default graph assignment.

2. The character string literal form of the node name is used as a value in the result of a query to record which subquery discovered the solution.

The character string literal form of the node name is defined in this section, the identifier form in the next. If i is a positive integer, let *lit*(i) be the shortest SQL literal for i. For example, ‘1’ is the shortest literal for 1, and ‘01’ is not. The SQL character string literal *NodeName*(DGA_i, $Node_j$) is

```
'D' || lit(i) || 'N' || lit(j)
```

For example, 'D1N1' is the node name of the root node with the first default graph assignment.

2.9 SQL identifiers

The generated SQL query will require a number of SQL identifiers. For this purpose, I define a function *Id* that generates SQL identifiers from pieces of SPARQL query text.

variables: rule [90] VARNAME is case sensitive, whereas SQL regular (unquoted) identifiers are not. I haven't checked, but I also believe some of the characters of VARNAME are not permitted in SQL regular identifiers. Fortunately, there is a workaround, quoted identifiers, which are surrounded by double quotes; any embedded double quotes are handled by repetition (thus """" is an SQL quoted identifier whose value is a single double quote). Let *Id* be a function that maps a VARNAME to the corresponding quoted SQL identifier.

blank node identifiers: I assume that the domain of *Id* also includes SPARQL text conforming to rule [70] BLANK_NODE_LABEL. The mapping is to a quoted identifier that begins with _: and suppresses any whitespace that might be present between _: and NCNAME. For example, *Id*(_:a) = "_:a".

table names: Every graph is named by an IRI. I assume *Id*(IRI) is an SQL identifier that names the table that stores the graph referenced by IRI.

query names: every default graph assignment DGA_i and node $Node_j$ and has a query name, called *Id*(DGA_i, $Node_j$). This is the SQL identifier whose spelling is the same as the value of *NodeName*(DGA_i, $Node_j$). For example, "D1N1" is the query name of the root node using the first default graph assignment.

2.10 Paths

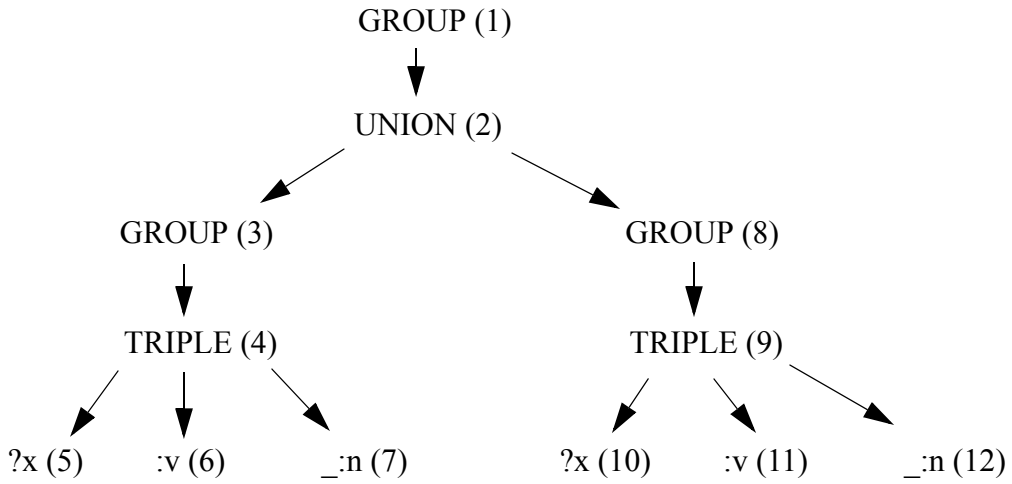
Every subquery will have as its first result column a column called "::path". This column will contain the concatenation of the character string form of every node name that figured in the generation of a solution.

The path column is used to provide the counting semantics that I have advocated. For example, given a pattern

```
{ { ?x :v _:n } UNION { ?x :v _:n } }
```

every solution of the first branch of the UNION is also a solution of the second branch. The SQL query generated by this proposal will distinguish duplicate solutions by means of the path column.

A solution that arises from the first branch will have a different path from a solution of the second branch. The tree for this example is



Node numbers are shown in parentheses. There is only one default graph assignment, since there is no GRAPH node. No subqueries are generated for the leaves of the tree. The first subqueries to be generated are the TRIPLE nodes, which have "::path" columns whose values are 'D1N4' and 'D1N9' respectively. The paths at their parents, the internal GROUP nodes, are 'D1N4D1N3' and 'D1N9D1N8' respectively. The UNION node forms the SQL UNION of these two streams, producing rows whose path is either 'D1N4D1N3D1N2' or 'D1N9D1N8D1N2'. Thus the history of each solution can be found in the "::path" column. This will enable us to use the SQL DISTINCT operator to factor out duplicates due to the blank node identifier _:n while retaining duplicates due to the UNION. At the very end, the path column is projected away.

2.11 Solution tables

The result of evaluating a subpattern is a table whose rows represent solutions of the subpattern. The first column of the table is always named "::path", and records the path through the tree that was used to derive a particular solution. The other columns are named after the bindables that appear in the subpattern. In the case of a constant subpattern, the only column is "::path"; the presence of a row indicates that the constant subpattern matched.

The value of a column other than "::path" may be null; this indicates that there is no binding for the corresponding bindable.

When one pattern is nested within another pattern, the larger pattern may contain bindables not found in the nested pattern. In that case, the SQL query for the nested pattern can be effectively extended with additional null columns for any bindables found in the containing pattern. The syntax to create such a column is

```
CAST (NULL AS CharType) AS Id(w)
```

where w is the bindable to be added to the solution table.

The algorithm constructs such widened queries as necessary. There is no conceptual difference between a row with a null value in a column, and a row that lacks that column altogether.

2.12 Glossary of symbols

symbol	meaning
g	number of named graphs
Tree	the tree representation of a SPARQL query
n	number of nodes in Tree
$Node_1$, ... $Node_n$	nodes of Tree
d	number of default graph assignments
DGA_1, ..., DGA_d	default graph assignments
NodeName	function to generate an SQL character string literal from a default graph assignment DGA_i and a node $Node_j$
Id	function to generate an SQL identifier from any of the following: 1. a bindable 2. a graph name IRI 3. a default graph assignment DGA_i and a node $Node_j$
v	a variable
w	a bindable
SQ	function that generates an SQL query for a particular default graph assignment DGA_i and a particular $Node_i$. The result of *SQ* is the solution table for the subpattern represented by $Node_i$ with default graph assignment DGA_i.

3. Transformation of nodes to SQL subqueries

This section is organized around the various kinds of nodes in a tree. For most nodes, we can treat the default graph assignment as a constant, DGA.

3.1 TRIPLE

If Node is a TRIPLE node of Tree, let the three children be $Node_{subject}$, $Node_{verb}$ and $Node_{object}$.

Form the character string:

```
NodeName(DGA, Node) AS "::path"
```

Let B be the set of bindables in $Node_{subject}$, $Node_{verb}$ and $Node_{object}$. For each element w of B, form the following strings:

if w is $Node_{subject}$, then

```
Subject AS Id(w)
```

else if w is $Node_{verb}$, then

```
Verb AS Id(w)
```

else

```
Object AS Id(w)
```

For example, the triple pattern ?x :v ?x only has one bindable, and only generates

```
Subject AS "x"
```

because the final "else" is never reached.

Let *TripleSelectList* consist of a comma-separated list of the preceding strings. Note that if there are no bindables, there is still a result column for "::path". The generated query in this case will return 1 row if the triple is found in the graph, otherwise it will return no rows. (The query can return at most one row because of the primary key on the Subject, Verb and Object columns of the graph's table.)

Define six predicates as follows:

If $Node_{subject}$ is an IRI or literal, let $Pred_{subject}$ be

```
Subject = Node_subject
```

If $Node_{verb}$ is an IRI or literal, let $Pred_{verb}$ be

```
Verb = Node_verb
```

If $Node_{object}$ is an IRI or literal, let $Pred_{object}$ be

```
Object = Node_object
```

If $Node_{subject}$ and $Node_{verb}$ are the same bindable, let $Pred_{sv}$ be

```
Subject = Verb
```

If $Node_{subject}$ and $Node_{object}$ are the same bindable, let $Pred_{so}$ be

```
Subject = Object
```

If $Node_{verb}$ and $Node_{object}$ are the same bindable, let $Pred_{vo}$ be

```
Verb = Object
```

Not all of these predicates will be defined. Let *TriplePred* be the logical conjunction of those that are defined (i.e., joined by AND). If none of the predicates is defined, let *Pred* be 1=1.

Then *SQ* (DGA, Node) is

```
Id(DGA,Node) AS
( SELECT TripleSelectList
  FROM Id (DGA(Node))
  WHERE TriplePred )
```

For example, node 4 above is the triple pattern

```
?x :v _:n
```

and is converted to

```
DIN4 AS
( SELECT 'D1N4' AS "::path", Subject AS "x", Object as "_:n"
  FROM GraphTable
  WHERE Verb = '//http:shenme.difang/v' )
```

assuming that the default graph is stored in GraphTable and the prefix ':' is mapped to 'http://shenme.difang/'

3.2 JOIN

If Node is a JOIN node of Tree, let its children be Ch_1, ..., Ch_c.

Form the character string

```
   Id(DGA,Chi)."::path"
|| ...
|| Id(DGA,Chc)."::path"
|| NodeName(DGA, Node) AS "::path"
```

For each bindable w appearing in any descendent of Node, let i be the lowest index such that w is in Ch_i. Form the strings:

```
Id(DGA,Chi).Id(w)
```

Let *JoinSelectList* be a comma-separated list of the preceding strings; if there is no bindable, there will still be at least a result column called "::path".

For every bindable w in any of the children, if w occurs in both Ch_i and Ch_j where i <> j, let *Pred*(w, i, j) be

```
(    Id(DGA,Chi).Id(w) IS NULL
  OR Id(DGA,Chj).Id(w) IS NULL
  OR Id(DGA,Chi).Id(w) = Id(DGA,Chj).Id(w) )
```

(That is, two solutions can join if they are equal on those bindables in the intersection of their domains.)

Let *JoinPred* be the logical conjunction of all defined *Pred* (w, i, j). If there are none, let *JoinPred* be

```
1=1
```

SQ(DGA, Node) is

```
Id(DGA, Node) AS
( SELECT JoinSelectList
  FROM Id(DGA,Ch1), . . ., Id(DGA,ChC)
  WHERE JoinPred )
```

Note that in the event that one of the children is a constant pattern that does not match, then the result above will also be empty, because the cross product in the FROM clause will be empty. If all the children are constant patterns that match, then the FROM clause will form the cross product of tables of 1 row each, resulting in a single row.

For example, consider the basic graph pattern

```
{ ?x :v1 _:n . _:n :v2 ?y }
```

The nodes in the tree are

$Node_1$: the root node, a GROUP node

$Node_2$: a JOIN node, the child of the root node

$Node_3$ and $Node_4$, the two TRIPLE nodes that are the children of $Node_2$.

There is only one default graph assignment. The queries for the two TRIPLE nodes are named D1N3 and D1N4. The query for the JOIN node is

```
D1N2 AS
( SELECT D1N3."::path" || D1N4."::path" || 'D1N2'
         AS "::path", D1N3."x", D1N3."_:n", D1N4."y"
  FROM D1N3, D1N4
  WHERE (    D1N3."_:n" IS NULL
          OR D1N4."_:n" IS NULL
          OR D1N3."_:n" = D1N4."_:n" )
```

This query can be considerably simplified (for example, it is deducible that D1N3."_:n" cannot be null because it must be the value of a primary key column of the graph table). The IS NULL conditions are necessary when joining the results of UNION or OPTIONAL nodes.

3.3 EMPTY

EMPTY is the node generated by an empty GroupGraphPattern, or one that contains only FILTERs. Let Node be an the EMPTY node. The SQL translation of an EMPTY node is

```
Id(DGA, Node) AS
( SELECT NodeName(DGA, Node) AS "::path"
  FROM VALUES (1) )
```

VALUES(1) is essentially a literal for a table with one row and one column, whose value is 1. If the SQL implementation does not support this feature of SQL, then any table known to have one row will do.

For example, if $Node_6$ is an EMPTY node and there is one default graph assignment, then we have the subquery

```
D1N6 AS
( SELECT 'D1N6' AS "::path"
  FROM VALUES (1) )
```

3.4 GROUP

Let Node be a GROUP node. A GROUP node corresponds to a GroupGraphPattern. In [SPARQL to trees], I followed the suggestion that a GROUP node is the appropriate place to evaluate FILTERs. The non-FILTER children of a GROUP node are handled the same as for a JOIN node. For simplicity, I assume that a GROUP has at most one non-FILTER child, which is a JOIN node or an EMPTY node. This can be assured by slightly modifying the algorithm in [SPARQL to trees], or by performing a preliminary pass over the tree to make it so. Let $Node_{join}$ be the JOIN or EMPTY node that is a child of GROUP.

Form the string

```
"::path" || NodeName(DGA,Node) AS "::path"
```

For each bindable w in the JOIN child of the GROUP node, form the string

```
Id(w)
```

A FILTER may contain bindables that are not found in $Node_{join}$. In that case, there will be no bindings for such bindables. This will be handled by widening the result of the JOIN child with additional null columns. If w is such a bindable, form the string

```
CAST (NULL AS CharType) AS Id(w)
```

Let *WidenSelectList* be a comma-separated list of the preceding strings. Let *WidenQuery* be

```
( SELECT WidenSelectList
  FROM Id(DGA, Node_join) ) AS W
```

Each FILTER node can be mapped to an SQL predicate. I have not worked out the complete details for this mapping, contenting myself with just a sketch here. The SQL boolean value *Unknown* (the same as the null value of the boolean type) corresponds to the error case in SPARQL. The mapping of the SPARQL predicate BOUND(w) is

```
(Id(w) IS NOT NULL)
```

Other SPARQL functions and operators are mapped to functions and operators in SQL. If the SQL function or operator F does not return null on null input, then this can be coerced by using the expression syntax

```
CASE WHEN a1 IS NULL OR ... OR an IS NULL THEN NULL
ELSE F(a1, ..., an) END
```

Let *FilterPred* be the logical conjunction of the mappings of all FILTER children of the GROUP node. If there are none, let *FilterPred* be 1=1.

SQ(DGA, Node) is

```
Id(DGA,Node) AS
( SELECT *
  FROM WidenQuery
  WHERE FilterPred )
```

For example,

```
{ ?s :v ?t FILTER ( LANG(?t) = 'en' ) }
```

This example has a single default graph assignment. Let the GROUP node be $Node_1$ and its child JOIN node be $Node_2$. Then we have

```
D1N1 AS
( SELECT *
  FROM ( SELECT "::path", "s", "t"
         FROM D1N2 ) AS W
  WHERE LANG ("t") = 'en' )
```

Here LANG is the SQL name of the function that implements the SPARQL function of the same name.

3.5 OPTIONAL

If Node is an OPTIONAL node of Tree, let its first child be $Node_{mand}$ (the mandatory pattern) and its second child be $Node_{supp}$ (the supplementary pattern).

Form the character string

```
   Id(DGA,Node_mand)."::path"
|| CASE WHEN Id(DGA,Node_supp)."::path" IS NULL
```

```
            THEN ''
            ELSE Id(DGA,Node_supp)."::path"
       END
  || NodeName(DGA,Node)
```

That is, the path of a solution from an OPTIONAL node includes the supplementary pattern's path only if the supplementary pattern contributed to the solution. The path always includes the mandatory pattern's path, as well as the node name of the OPTIONAL node itself.

For each bindable w that is in both the mandatory pattern and the supplementary pattern, form the string

```
COALESCE (Id(DGA,Node_mand).Id(w), Id(DGA,Node_supp).Id(w))
AS Id(w)
```

(COALESCE is an SQL operator whose result is the first operand if non-null, otherwise the second operand.)

For each bindable w in the mandatory pattern but not in the supplementary pattern, form the string

```
Id(DGA,Node_mand).Id(w)
```

For each bindable w in the supplementary pattern but not in the mandatory pattern, form the string

```
Id(DGA,Node_supp).Id(w)
```

Let *OptionalSelectList* be a comma-separated list of the preceding strings.

For each bindable w that is in both the mandatory pattern and the supplementary pattern, form the string

```
(    Id(DGA,Node_mand).Id(w) IS NULL
  OR Id(DGA,Node_supp).Id(w) IS NULL
  OR Id(DGA,Node_mand).Id(w) = Id(DGA,Node_supp).Id(w) )
```

Let *OptionalPred* be the logical conjunction of the preceding predicates; if there are none, let *OptionalPred* be 1=1.

SQ(DGA, Node) is

```
Id(DGA,Node) AS
( SELECT OptionalSelectList
  FROM Id(DGA,Node_mand) LEFT OUTER JOIN Id(DGA,Node_supp)
       ON OptionalPred )
```

For example, if we change the example of a JOIN to have an OPTIONAL:

```
{ ?x :v1 _:n OPTIONAL { _:n :v2 ?y } }
```

The tree has the following nodes:

$Node_1$: the root node, a GROUP node

$Node_2$: the OPTIONAL node

$Node_3$: the mandatory pattern, a TRIPLE node

$Node_4$: the supplementary pattern, a GROUP node

$Node_5$: the child of $Node_4$, a TRIPLE node.

There is only one default graph assignment. The queries for the two operands of OPTIONAL are named D1N3 and D1N4. The query for the OPTIONAL node is

```
D1N2 AS
( SELECT    D1N3."::path"
         || CASE WHEN D1N4."::path" IS NULL THEN ''
                 ELSE D1N4."::path" END
         || 'D1N2' AS "::path",
         D1N3."x",
         COALESCE (D1N3."_:n", D1N4."_:n") AS "_:n",
         D1N4."y"
  FROM D1N3 LEFT OUTER JOIN D1N4
    ON (    D1N3."_:n" IS NULL
         OR D1N4."_:n" IS NULL
         OR D1N3."_:n" = D1N4."_:n" ) )
```

3.6 UNION

If Node is a UNION node of Tree, let its children be Ch_1, ..., Ch_c.

Let the bindables that appear in Node be B. Arrange these bindables in some fixed order, w_1, . . ., w_b.

For each i between 1 and c, construct *UnionSelectList*$_i$ as a comma-separated list whose components are:

"::path" || *NodeName*(DGA,Ch_i) AS "::path"

and

for each j between 1 and b,

if w_j appears in Ch_i, then

Id(w_j)

else

CAST (NULL AS *CharType*) AS *Id*(w_j)

SQ(DGA, Node) is

```
Id(DGA, Node) AS
( SELECT UnionSelectList1
  FROM Id(DGA,Ch1)
  UNION ALL
  . . .
  SELECT UnionSelectListc
  FROM Id(DGA,Chc)
)
```

For example

```
{ { ?x :v1 _:n } UNION { _:n :v2 ?y } }
```

has the following nodes:

$Node_1$: the root GROUP node

$Node_2$: the UNION node

$Node_3$: the first operand of UNION, a GROUP node

$Node_4$: the TRIPLE beneath $Node_3$

$Node_5$, $Node_6$, $Node_7$: the leaves beneath $Node_4$

$Node_8$: the second operand of UNION, another GROUP node

$Node_9$: the TRIPLE beneath $Node_8$

$Node_{10}$, $Node_{11}$, $Node_{12}$: the leaves beneath $Node_9$

There is only one default graph assignment. The inputs to the UNION are the subqueries named D1N3 and D1N8. Let the bindables in this query be placed in the order ?x, _:n, ?y . The query for the UNION is

```
D1N2 AS
( SELECT "::path" || 'D1N3' AS "::path",
         "x", "_:n", CAST (NULL AS CLOB) AS "y"
  FROM D1N3
  UNION ALL
  SELECT "::path" || 'D1N8' AS "::path",
         CAST (NULL AS CLOB) AS "x", "_:n", "y"
  FROM D1N8
)
```

using CLOB as the character type for Subject, Verb and Object columns.

3.7 GRAPH

Let Node be a GRAPH node. Node has two children, the graph selector and a pattern. The second child will always be a GROUP node; let it be $Node_{group}$.

I do not believe it will be possible to extend the entailment technique of pattern matching to permit the scope of blank node identifiers to cross GRAPH nodes. See [Constructive mapping semantics] section 3 "Entailment reconsidered". Consequently I propose that the solutions of a GRAPH node must be reduced by factoring out blank node identifiers.

Let *L* be an SQL character string literal whose value is $DGA_i(Node_{group})$, that is, the IRI that is chosen by the default graph assignment for the GROUP node beneath the GRAPH node.

Form the character string

```
"::path" || NodeName(DGA,Node) AS "::path"
```

For every variable v in $Node_{group}$, form the character string

```
Id(v)
```

If the graph selector s is a bindable not found in $Node_{group}$, form the character string

```
L AS Id(s)
```

Let *GraphSelectList* be a comma-separated list of the preceding character strings.

If the graph selector s is a variable and appears in $Node_{group}$, let *GraphPred* be

```
Id(s) = L
```

otherwise let *GraphPred* be

```
1=1
```

then *SQ*(DGA, Node) is

```
Id(DGA, Node) AS
( SELECT DISTINCT GraphSelectList
  FROM Id(DGA, Node_group)
  WHERE GraphPred
)
```

For example, suppose there are two named graphs, with IRIs http://shenme.mingzi and http://nin.qui.xing . The SPARQL query

```
{ ?g GRAPH { ?s :v _:n } }
```

is graphed with the following nodes:

$Node_1$: the root node, a GROUP node

$Node_2$: the GRAPH node

$Node_3$: the graph selector node, ?g

$Node_4$: the second child of the GRAPH node, a GROUP node

$Node_5$: the TRIPLE node beneath $Node_4$

$Node_6$, $Node_7$, $Node_8$: the leaves beneath $Node_5$

There are two default graph assignments.

DGA_1 assigns the default graph to $Node_1$ and $Node_2$, and assigns http://shenme.mingzi to the other nodes

DGA_2 assigns the default graph to $Node_1$ and $Node_2$, and assigns http://nin.gui.xing to the other nodes

There are two queries generated for every node of the graph. At the GRAPH node, we have these two queries:

```
D1N2 AS
( SELECT DISTINCT "::path" || 'D1N2' AS "::path",
         'http://shenme.mingzi' AS "g", "s"
  FROM D1N4
  WHERE 1=1 )
```

and

```
D2N2 AS
( SELECT DISTINCT "::path" || 'D2N2' AS "::path",
         'http://nin.gui.xing' AS "g", "s"
  FROM D2N4
  WHERE 1=1 )
```

Note that the constraint that the pattern `{ ?s :v _:n }` must be matched within a particular graph is handled within the previously defined queries D1N4 and D2N4. These queries are essentially TRIPLE nodes, and the FROM clause in each query will reference the appropriate table based on the default graph assignment, DGA_1 in the case of D1N4 and DGA_2 within D2N4.

Minor issue: this does not assure that the value of ?g is in the default graph. I am not sure whether it is intended that in a query such as

```
{ ?g GRAPH { ?s :v _:n } }
```

that the value of ?g must match some node of the default graph. My preference is that if ?g does not appear in a triple pattern of the default graph, then the scoping set for ?g should be the set of graph names of named graphs, rather than the set of terms of the default graph.

3.8 Root node

Let $Node_1$ be the root node of Tree. $Node_1$ is a GROUP node. Preceding rules have defined *SQ*(DGA_i, $Node_j$) for all default graph assignments DGA_i and all nodes $Node_j$, including in particular *SQ*(DGA_i, $Node_1$).

For each variable v appearing anywhere in Tree, form the character string

```
Id(v)
```

Let *FinalSelectList* be a comma separated list of these strings.

Let *AlmostFinalSelectList* be the following:

```
"::path", FinalSelectList
```

The purpose of *AlmostFinalSelectList* is to project away bindings to blank node identifiers. *AlmostFinalSelectList* is executed with SELECT DISTINCT, so distinctions based on the path are retained, but distinctions between rows based only on bindings to blank node identifiers are purged.

The purpose of *FinalSelectList* is to project away the "::path" column, which is generated solely for bookkeeping. *FinalSelectList* is not performed with DISTINCT, so that solutions that are found along different paths are retained. This provides the counting semantics that I have advocated.

For each default graph assignment DGA_i, let $WithClause_i$ be a comma-separated list of *SQ*(DGA_i, $Node_j$) for all nodes $Node_j$, in descending order by j. This ordering insures that each named subquery is defined before it is referenced.

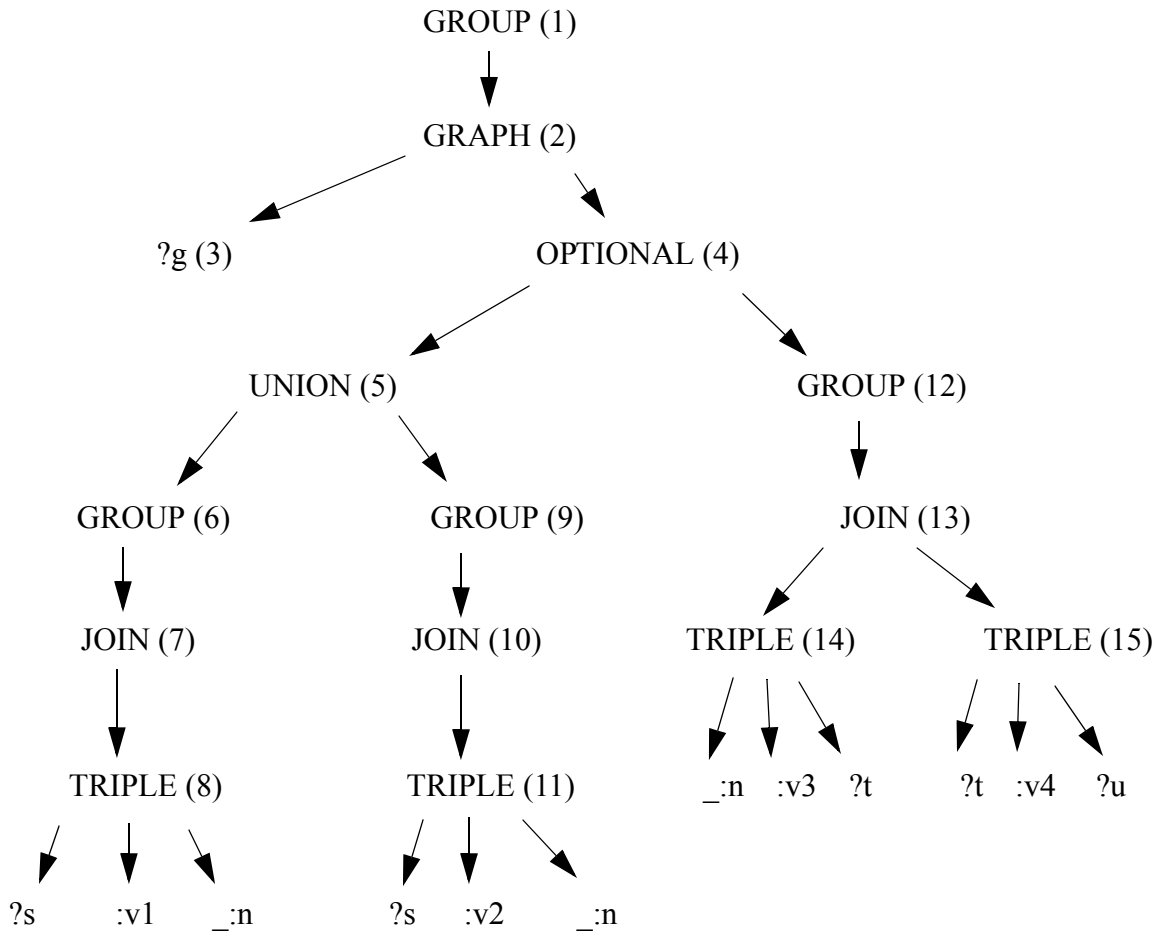
The SQL query that solves the Tree as a whole is

```
WITH WithClause1, . . ., WithClaused
SELECT FinalSelectList
FROM (
       SELECT DISTINCT AlmostFinalSelectList
       FROM Id(DGA1, Node1)
     UNION ALL
       SELECT DISTINCT AlmostFinalSelectList
       FROM Id(DGA2, Node1)
     UNION ALL
       . . .
       SELECT DISTINCT AlmostFinalSelectList
       FROM Id(DGAd, Node1)
```

Complete example:

```
{ ?g GRAPH { { ?s :v1 _:n } UNION { ?s :v2 _:n }
             OPTIONAL { _:n :v3 ?t . ?t :v4 ?u } } }
```

Tree:



Suppose there are two named graphs, which will cause two default graph assignments, which differ at node 4 and below. Let Table1 and Table2 be the tables that store the two named graphs. Let http://shenme.difang/ be the expansion of the prefix : .

```
WITH
D1N15 AS /* TRIPLE { ?t :v4 ?u } */
         ( SELECT 'D1N15' AS "::path",
                  Subject AS "t", Object AS "u"
           FROM Table1
           WHERE Verb = 'http://shenme.difang/v4' ),
D1N14 AS /*  TRIPLE { _:n :v3 ?t } */
        ( SELECT 'D1N14' AS "::path",
                  Subject AS "_:n", Object AS "t"
           FROM Table1
           WHERE Verb = 'http://shenme.difang/v3' ),
D1N13 AS /* JOIN (DIN14, D1N15) */
```

```
          (  SELECT  D1N14."::path" || DIN15."path"
                    ||'D1N13' AS "::path",
                  D1N14."_:n", D1N14."t", D1N15."u"
            FROM D1N14, D1N15
            WHERE ( D1N14."t" IS NULL OR D1N15."t" IS NULL
                    OR D1N14."t" = D1N15."t" )
D1N12 AS /* GROUP (D1N13) */
          ( SELECT *
            FROM  ( SELECT "::path" || 'D1N12' AS "::path",
                   "_:n", "t", "u"
                    FROM D1N13 ) AS W
            WHERE 1=1 ),
D1N11 AS /* TRIPLE { ?s :v2 _:n } */
          ( SELECT 'D1N15' AS "::path",
                    Subject AS "s", Object AS "_:n"
            FROM Table1
            WHERE Verb = 'http://shenme.difang/v2' ),
D1N10 AS /* JOIN (D1N11) */
          ( SELECT D1N11."::path" || 'D1N10'  AS "::path",
                   D1N11."s", D1N11."_:n"
            FROM D1N11
            WHERE 1=1 ),
D1N9  AS /* GROUP (D1N10) */
          ( SELECT *
            FROM  ( SELECT "::path" || 'D1N10' AS "::path",
                   "s", "_:n"
                    FROM D1N10 ) AS W
            WHERE 1=1 ),
D1N8  AS /* TRIPLE { ?s :v1 _:n } */
          ( SELECT 'D1N8' AS "::path",
                    Subject AS "s", Object AS "_:n"
            FROM Table1
            WHERE Verb = 'http://shenme.difang/v1' ),
D1N7  AS /* JOIN (D1N8) */
          ( SELECT D1N8."::path" || 'D1N7'  AS "::path",
                  D1N8."s", D1N8."_:n"
            FROM D1N8
            WHERE 1=1 ),
D1N6  AS /* GROUP (D1N7) */
          ( SELECT *
            FROM  ( SELECT "::path" || 'D1N10' AS "::path",
                   "s", "_:n"
                    FROM D1N9 ) AS W
            WHERE 1=1 ),
D1N5   AS /* UNION (D1N6, D1N9) */
```

```
          ( SELECT "::path" || 'D1N5' AS "::path",
                   "s", "_:n"
            FROM D1N6
            UNION ALL
            SELECT "::path" || 'D1N5' AS "::path",
                   "s", "_:n"
            FROM D1N9 )
D1N4  AS /* OPTIONAL (D1N5, D1N12) */
         ( SELECT D1N5."::path" ||
                  CASE WHEN D1N12."::path" IS NULL THEN ''
                       ELSE D1N12."::path" END
                  || 'D1N4' AS "::path",
                  D1N5."s",
                  COALESCE(D1N5."_:w",D1N12."_:w") AS "_:w",
                  D1N12."t", D1N12."u"
           FROM D1N5 LEFT OUTER JOIN D1N12
                ON (   D1N5."_:n" IS NULL
                    OR D1N12."_:n" IS NULL
                    OR D1N5."_:n" = D1N12."_:n") )
/* no subquery for D1N3, it is just a variable node */
D1N2  AS /* GRAPH (?g, D1N4) */
         ( SELECT DISTINCT "::path" || 'D1N2' AS "::path",
                           'http://shenme.mingzi' AS "g",
                           "s", "t", "u",
           FROM D1N4
           WHERE 1=1 ),
D1N1 AS /* GROUP (D1N2) */
        ( SELECT *
          FROM ( SELECT "::path" || 'D1N1' AS "::path",
                        "g", "s", "t", "u"
                 FROM D1N2 ) AS W
          WHERE 1=1 )
/*
** plus subqueries D2N15 through D2N1 as clones of the
** above, with D1 changed to D2, and Table1 changed to
** Table2
*/
SELECT "g", "s", "t", "u"
FROM ( SELECT DISTINCT "::path", "g", "s", "t", "u"
       FROM D1N1
       UNION ALL
       SELECT DISTINCT "::path", "g", "s", "t", "u"
       FROM D2N1 )
```

- End of paper -