A Mapping of SPARQL Onto Conventional SQL

Eric Prud’hommeaux, Alexandre Bertails
World Wide Web Consortium (W3C)
{eric,bertails}@w3.org
http://www.w3.org/

Abstract. This paper documents a semantics for expressing relational data as an RDF graph [RDF] and an algebra for mapping SPARQL SELECT queries over that RDF to SQL queries over the original relational data. The RDF graph, called the stem graph, is constructed from the relational structure and a URI identifier called a stem URI. The algebra specifies a function taking a stem URI, a relational schema and a SPARQL query over the corresponding stem graph, and emitting a relational query, which, when executed over the relational data, produces the same solutions at the SPARQL query executed over the stem graph. Relational databases exposed on the Semantic Web can be queried with SPARQL with the same performance as with SQL.

Key words: Semantic Web, SPARQL, SQL, RDF

1 Introduction

Motivations to bridge SPARQL and SQL abound: science, technology and business need to integrate more and more diverse data sources; the far majority of data that we expect machines to interpret are in relational databases; SQL federation extensions like SchemaSQL [SSQL] don’t place these databases in the more expressive Semantic Web; etc. These points have motivated many developers to produce mapping tools, but robustness and performance have been a problem, in part because of a lack of a standard semantics for this mapping. In order to draw the desired investment in the Semantic Web, we need to match the performance of conventional relational databases.

The semantics of a SPARQL query are specified directly in the SPARQL Specification [SPARQL], while the semantics for SQL can be expressed in Extended three-valued Predicate Calculus (E3VPC) [FSQL]. We draw on the body of mappings from SPARQL to SQL triple stores to develop a mapping to conventional relational stores, which are more efficient and compact than triple stores. The crux of this mapping is that a SPARQL Basic Graph Pattern (BGP) can be broken down into a set of conjunctions of triple patterns, where each conjunct matches the attributes of a given relation in the database. These conjuncts can be expressed as a single relvar (see Relational Terminology below). An arbitrary SPARQL query, with conjunctions, disjunctions and optionals, can be expressed as a single SQL query and performed with the same efficiency as a conventional SQL query.
Creating a computable mapping from the SPARQL semantics to SQL semantics provides a SPARQL tool vendors with an implementable and testable specification, and SPARQL users with a sound and complete foundation for the next level of the Semantic Web. It also frees database custodians of a false choice between a mature but insular database technology and a promising but naive newcomer.

2 Relational Terminology

Our examples reference this example table\(^1\). As it is not our goal to provide coverage of XML or SQL datatypes, it includes only primary and foreign keys (as integers), strings, and dates.

<table>
<thead>
<tr>
<th>Table Employee</th>
<th>empid</th>
<th>lastName</th>
<th>birthday</th>
<th>manager</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Johnson</td>
<td>1969-11-08</td>
<td>NULL</td>
<td>tools</td>
<td></td>
</tr>
<tr>
<td>253</td>
<td>Smith</td>
<td>1979-01-18</td>
<td>18</td>
<td>tools</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td>Jones</td>
<td>1981-03-24</td>
<td>253</td>
<td>tools</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Xu</td>
<td>1966-11-08</td>
<td>NULL</td>
<td>toys</td>
<td></td>
</tr>
<tr>
<td>254</td>
<td>Ishita</td>
<td>1971-10-31</td>
<td>253</td>
<td>toys</td>
<td></td>
</tr>
</tbody>
</table>

A database is a set of relations in a given schema. The schema for each relation defines a heading, set of attributes, each of some datatype. If the attribute is a foreign key, the attribute’s target is another attribute in some relation. If the attribute is not a foreign key, the attribute is a primitive attribute and the attribute’s datatype is the datatype of all values of that attribute (e.g. CHAR(40), INTEGER).

As terminology around relational databases varies, we will borrow from one of the simpler definitions. As the expressivity of SPARQL is limited compared to SQL, we need only a subset of the SQL semantics. Borrowing an SQL terminology from the semantics review in [SSQL], we parse an SQL query into:

```sql
SELECT attrList FROM fromList WHERE whereCondition
```

The fromList is a list of tuples mapping relations to variables called relvars. The relvar identifies a multiset of tuples, the tuples from the relation. Conceptually, an aggregate relvar is formed by algebraic combination of the relvars in the fromList. This mapping requires only the two algebraic operators INNER JOIN and LEFT OUTER JOIN, which are indicated by keywords separating tuples in the fromList. The whereCondition is a constraint expressed in terms of the attributes of the relvars (relvarattrs) in the fromList. The whereCondition includes the conjunction of the expressions connecting foreign keys to primary keys (e.g. `employee.address=empAddress.id`). The aggregate relvar is restricted to those

---

\(^1\) there is only one table, in part to keep the reader’s context at a minimum, and in part to exercise the appropriate naming for relvars (FROM Employee AS manager).
tuples passing the the whereCondition constraint. The attrList selects a set of attribute from the restricted aggregate relvar and renames them to new attribute names, producing a new relation in the schema defined by the attrList.

3 Stem Graph

While there are many applications where a specific stem graph can be mapped to a non-RDF data model or query language, the most obvious is probably relational. There are many possible expressions of relational data in RDF; the following one requires as input, a stem URI, a database, and some miscellaneous punctuation for separating components in constructed URIs. In effect, the stem URI creates a web space for the records in the database.

The relation stem graph is composed by applying a relation map to each of the tuples in a relation. The stem graph is the union of all of the relation stem graphs. A relation map defines an RDF triple [RDF] for each attribute of a non-null tuple in a relation. For any tuple, the relation map emits triples with the same subject [RDF]. The subject is a URI formed by a node map of the primary key, if the relation has a primary key, or by a novel blank node [RDF] otherwise. For each attribute, the corresponding triple has this subject, a predicate [RDF] formed by a predicate map, and an object [RDF] formed by a node map of the referenced primary key if it is a foreign key, or by a literal map otherwise.

A predicate map is a mapping function which emits a URI derived by concatenating the stem URI, relation name, and the attribute name, separated by punctuation discussed below. The (infinite) set of URIs which are produced by a predicate map given a particular relation are said to be in that relation. A predicate unmap is the reverse function, parsing a URI in a relation to derive the stem URI, relation name and attribute name. These are called the predicate’s stem, predicate’s relation, and predicate’s attribute respectively. If the predicate’s attribute is a foreign key, the predicate’s target is the attribute’s target. If the predicate’s attribute is not a foreign key, the predicate’s datatype is the datatype of the predicate’s attribute.

A node map is a mapping function emits a URI derived by concatenating the stem URI, the primary key name, and the primary key value, separated by punctuation discussed below. The set of URIs emitted by a node map over a relation are in that relation. A node unmap function derives the stem URI, relation name, primary key name and primary key value of a URI in a relation. These are called the node’s stem, node’s relation, node’s attribute, and node’s key value respectively.

The node and predicate maps require some separators to facilitate unmapping. These separators are chosen to be compatible with linked data, that is, the cloud of resources which would be in the same web page is of a reasonable size and practical comprehension.

subject map: stem ‘/’ table ‘/’ attribute ‘.’ value # fragment
predicate map: stem ‘/’ table ‘#’ attribute

2 for simplicity, this treatment does not handle compound primary keys.
RDF literals with no datatype are called RDF Plain literals [RDF]. A literal map is a mapping function which emits an RDF literal [RDF] with a lexical form equivalent to the lexical value in the tuple, and a datatype selected from the mapping below:

<table>
<thead>
<tr>
<th>SQL datatype</th>
<th>XML Schema datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>xsd:string</td>
</tr>
<tr>
<td>VARCHAR(u)</td>
<td>xsd:string</td>
</tr>
<tr>
<td>INT</td>
<td>xsd:integer</td>
</tr>
<tr>
<td>DATE</td>
<td>xsd:date</td>
</tr>
</tbody>
</table>

A literal unmap maps an RDF literal to a SQL term.

As an example, we can express the stem map of following table a as turtle, using http://hr.example/DB/ as the stem URL:

```
Table Employee
    id   lastName   manager
    18   Johnson    NULL
    253  Smith      18

@prefix emplP: <http://hr.example/DB/Employee#> .
@prefix emplN: <http://hr.example/DB/Employee/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
emplN:id.18 emplP:lastName "Johnson"^^xsd:string .
emplN:id.253 emplP:lastName "Smith"^^xsd:string .
emplN:id.253 emplP:manager emplN:id.18 .
```

3.1 Literal Maps

The above relational strings (or varschars, or however the last names might be encoded) are expressed as RDF literals. For Common relational datatypes, have corresponding W3C XML Schema (XSD) datatypes, which are available for a literal map, e.g. a mapping from a relational date to an xsd:date:

```
Table Employee
    id   ... birthday
    18   ... 1969-11-08
    253  ... 1979-01-18
```

\[ it \ is \ expected \ that \ the \ upcoming \ ISO \ SQL \ specification \ will \ include \ a \ bidirectional \ mapping \ between \ SQL \ datatypes \ and \ XML \ Schema \ datatypes, \ including \ (e.g. \ numeric) \ canonicalization \ requirements, \ and \ mapping \ between \ SQL \ character \ sets \ and \ Unicode \ code \ points. \]
The literal mapping used in examples in this paper is:

<table>
<thead>
<tr>
<th>relational type SQL</th>
<th>RDF type SPARQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>literal</td>
</tr>
<tr>
<td>integer</td>
<td>xsd:integer</td>
</tr>
<tr>
<td>float</td>
<td>xsd:float</td>
</tr>
<tr>
<td>date</td>
<td>xsd:date</td>
</tr>
</tbody>
</table>

The core SPARQL language demands only a subset of these XSD datatypes, however, support for additional datatypes is well-defined, and trivial to implement by pushing evaluation into the SQL query. For example, a SPARQL constraint \(?bday=\"1969-11-08\"\) is easy to convert to an SQL constraint \(manager.birthday=\"1969-11-08\"\).

4 Stem Query as SQL

Informally, a stem query is a SPARQL query which matches some possible stem graph created from a database. The triple patterns and FILTERs must individually and collectively respect the relational schema. Following is a more formal definition of a stem query.

A node var is a variable in a triple pattern in place of a node, e.g. emplN:id.18#record in the above example. The node var’s relation is the predicate’s relation if the node var is in the subject position, or the predicate’s target if the node var is in the object position of a foreign key predicate.

A primitive var is a variable in the object position of a triple pattern whose predicate is a primitive attribute. The primitive var’s datatype is the datatype of the predicate’s attribute.

A triple pattern is in S if it meets all of these conditions:

- The predicate is in some relation R in the stem graph.
- The subject is a variable or is a node in the predicate’s relation.
- The object is a variable or is a node in the predicate’s target relation (if the predicate’s attribute is a foreign key) or is a literal with a datatype matching the predicate’s datatype.

A SPARQL query is a stem query for a relation schema S if it meets the following conditions:

- Every triple pattern is in S.
- Every node var for a given variable is for the same relation.
- Every primitive var for a given variable has the same datatype.
Given the mapping from relational data to stem graphs, it is possible to convert stem queries to relational queries. A mapped stem query is a relational query that, when applied to any possible relational data in a given schema, produces the same result set as would the stem query performed on the stem graph for that data. Stem queries can be expressed as SQL, or as an execution plan in a relational database. The following sections examine the logical components of SPARQL queries and illustrate how they are mapped to SQL.

The function \texttt{mapGraphPattern} maps a SPARQL graph pattern (one of \texttt{BasicGraphPattern}, \texttt{Filter}, \texttt{GroupGraphPattern}, \texttt{UnionGraphPattern}, \texttt{OptionalGraphPattern}, but not \texttt{GraphGraphPattern}) to a \texttt{graphPatternMappingState} composed of a \texttt{tableJoinList}, \texttt{varmap}, and a \texttt{graphPatternExpression}. The \texttt{tableJoinList} is a set of joined or outer joined relvars. Because the order of outer joins and joined \texttt{UNION}s affects semantics, the \texttt{tableJoinList} preserves the order of the first time a table is inserted into the list. A \texttt{varmap} is a map of SPARQL variable to relvarattr for the variables in that graph pattern. The \texttt{graphPatternExpression} expresses the SQL constraints representing SPARQL \texttt{FILTER}s and the coreferences of SPARQL variables within the graph patterns.

Applying \texttt{mapGraphPattern} to a BasicGraphPattern breaks that BGP into smaller patterns and aggregates their graphPatternMappingStates into a single state. The same aggregation is applied to produce a single graphPatternMappingState for all the graph patterns in a GroupGraphPattern.

### 4.1 Basic Graph Patterns

A Basic Graph Pattern is set of triple patterns which have similar expressivity to their SQL counterparts, joins and restrictions. Just as tuple maps produce triples from tuple attributes, the process can be reversed, converting triple patterns to queries on tuples.

Taking advantage of the SPARQL join semantics, we can sequester the triple patterns in a BGP by subject, assign each a relvar, and emulate the SPARQL join in SQL by adding constraints that none of the predicate’s attributes is non-null\(^4\). The SPARQL definition in terms of “distinct RDF instance mappings” [SPARQL] is met by the cross product (\texttt{INNER JOIN}) of the aggregate tuple, restricted by a set of constraints coming from variable coreferences in the graph pattern.

In this SPARQL query to select the names of employees and their managers,

\begin{verbatim}
PREFIX emplP: <http://hr.example/DB/Employee#>
SELECT ?empName ?managName
WHERE { ?emp emplP:lastName ?empName .
  ?manager emplP:lastName ?managName }
\end{verbatim}

\(^4\) Variable predicates are not treated in this document, though an explicit enumeration of table attributes would probably suffice.
the third triple pattern is on the same relation (Employee), but is intended to match managers, rather than their employees. This can be expressed using different table aliases for each tuple map with the same table and different subject. For readability in this document, these table names are derived from the variable name in the subject of the triple pattern:

```sql
SELECT emp.lastName, manager.lastName
FROM Employee as emp
    INNER JOIN Employee as manager ON emp.manager=manager.empid
WHERE emp.lastName IS NOT NULL AND manager.lastName IS NOT NULL
```

This BGP has a coreference on `?manager`; it is used as the object of the second triple pattern, and the subject of the third. This coreference is enforced in the join constraint `emp.manager=manager.empid`.

The triple patterns in the BGP are grouped by subject. Each group is assigned a relvar with a name unique within the query and the schema. The examples in the document generates a relvar name by a lexical trasformation of the subject, where subject variables produce a relvar with the same name as the variable, and subject URIs produce a relvar with a name composed of the key name and value which were unmapped from the URI.

Examining each triple in a subject BGP, a variable in the subject binds that variable to relvarattr composed of the relvar name and the relations’s primary key. A variable in the object position binds that variable to a relvarattr composed of the relvar name and the predicate’s attribute (e.g. emp.lastName). Adding a variable mapping to a varmap which already maps that variable results in a varmap with only one of those bindings (the first binding in our examples) and a coreference constraint of the form `relvarattr1=relvarattr2`, e.g. `emp.manager=manager.id` in the above example.

4.2 Aggregation

The result of a mapGraphPattern on a BGP consists of a tableList, a varmap and an expression. The tableList is the set of relvars for each subject bgp. The varmap is the union of the varmaps of the subject bgps, where coreferences produce a coreference constraint. This process is used to aggregate any graphPatternMappingStates into a singel state, for instance, to map a query with a BasicGraphPattern followed by a \textbf{UNION}.

4.3 Non-NULL Constraints

Each triple pattern in a SPARQL basic graph pattern must match a triple in the stem graph. \textbf{NULL} attributes in the relational database prevent the expression of that triple in the graph pattern. Therefore, we must take care to prevent SQL

\footnote{any triple patterns sharing a subject must have a predicate in the same relation in order to match the stem graph.}
from matching NULL attributes in mapped stem query. mapGraphPattern adds to the resulting expression a set of IS NOT NULL constraints for each variable in the basic graph pattern. As an optimization, you can elide any NOT NULL constraints for variables which are used in the expression which would fail with NULL arguments (e.g. =, !=, <, AND, ...).

4.4 Parsing Node Identifiers

Queries may include URLs in the subject position, or in the object position of predicates whose attribute is a foreign key. Any node identifiers in sequestered BGP's produce a Node ID constraint, which is a conjunct in the whereCondition.

A Node ID constraint in the subject position introduces a constraint that the predicate's primary key is equal to the node's key value derived from a node unmap. A Node ID constraint in the object position introduces a constraint that the predicate's primary target is equal to the node's key value derived from a node unmap.

This query lists the names of the employees who work for Employee 18:

```sql
PREFIX emplP: <http://hr.example/DB/Employee#>
PREFIX emplN: <http://hr.example/DB/Employee/>
SELECT ?empName
WHERE { ?emp emplP:lastName ?empName .
  ?emp emplP:manager emplN:id.18 } 
```

The inclusion of a foreign key (manager) introduces a join, and the value of that foreign key (18) produces a constraint `manager=18`. The query would be inconsistent if `http://hr.example/DB/Employee#manager` and `http://hr.example/DB/Employee/id.18` did not both refer to the same table.

```sql
SELECT emp.lastName
FROM Employee as emp
WHERE emp.manager=18 AND emp.lastName IS NOT NULL 
```

If `emplN:id.18` were replaced with a variable, say `?manager`, the SQL would need a join `Employee AS manager` only if `?manager` were selected or if some property of manager were referenced, e.g. `?manager emplP:lastName ?manName`.

4.5 Filters on Basic Graph Patterns

SQL filters and SPARQL filters have similar arithmetic and boolean evaluation, e.g. `manager.birthday > "1969-01-01"^^xsd:date` is expressed in SQL as `manager.birthday > "1969-01-01"`.

6 OPTIONAL patterns permitting the optional matching of entire graph patterns is discussed below in Optionals.

7 This document does not discuss the edges of datatype compatibility (e.g. maximum integer or float value) as this would require a survey of the SQL implementations.
A query for third-line employees include multiple joins. A constraint on the second-line manager can constrain the join of that table. For example, in this query with many joins:

```
PREFIX emplP: <http://hr.example/DB/Employee#>
SELECT ?empName ?grManName
WHERE { ?emp emplP:lastName ?empName .
?emp emplP:birthday ?empBday .
?manager emplP:birthday ?manBday .
?grandMan emplP:birthday ?grandManBday .
?grandMan emplP:lastName ?grManName
FILTER (?manBday<?empBday && ?grandManBday<?manBday) }
```

The filter constraint conjuncts (?manBday<?empBday and ?grandManBday<?manBday), can be sorted into the join constraints:

```
SELECT emp.lastName AS empName, grandMan.lastName AS grManName
FROM Employee AS emp
  INNER JOIN Employee AS manager ON manager.empid=emp.manager
  INNER JOIN Employee AS grandMan
    ON grandMan.empid=manager.manager
WHERE grandMan.birthday<manager.birthday
  AND manager.birthday<emp.birthday
  AND emp.lastName IS NOT NULL AND grandMan.lastName IS NOT NULL
```

Optionals and Disjunctions can produce SQL subqueries. Filters on these subqueries should be treated as applying to the ON constraint when the subquery in joined to the rest of the query. Constants and variables introduced in an OPTIONAL or UNION may be pushed down into the subquery when generating SQL strings. When generating execution plans which exceed SQL expressivity, it may be possible to push into the subquery constraints against variables bound outside of the subquery.

### 4.6 Disjunctions

SPARQL UNIONS can share variables with their surround context (for example, ?who and ?bday below). The disjoints in a UNION may bind different variables (?bday is only referenced in the second disjoint below).

```
SELECT ?name
WHERE { ?who emplP:name "Smith"
    ?manager emplP:lastName ?name }
  UNION
```
The mapped stem query can invent relvarattrs (union1._DISJOINT_ below) to identify which disjoint bound any tuple in the union. Constraints can use these relvarattrs to conditionally unify with relvarattrs bound elsewhere in the query.

```sql
SELECT union1.name
FROM Employee AS who
INNER JOIN (SELECT 0 AS _DISJOINT_, manager.lastName AS name,
               manager.manages AS who, NULL AS bday FROM Employee as manager
               WHERE manager.lastName IS NOT NULL AND manager.lastName IS NOT NULL
               UNION
               SELECT 1 AS _DISJOINT_, managed.lastName AS name, who.empid AS who,
               managed.birthday AS bday FROM Employee AS who
               INNER JOIN Employee as managed ON below.manages=managed.empid
               WHERE managed.lastName IS NOT NULL
           ) AS union1 ON (_DISJOINT_=0 AND union1.who=who.empid) OR
             (_DISJOINT_=1 AND union1.who=who.empid AND union1.bday=who.birthday)
WHERE who.lastName="Smith"
```

The union is given a unique relvar name, union1 above. Relational theory asserts that the attrlists in each of the subselects in a UNION need to have the same semantics (and datatype). This is ensured by ordering the selected variables from unionVars, a list extracted from the set of all variables in all of the disjoints. The disjoints are numbered, and each is processed by mapGraphPattern, producing a graphPatternMappingState called the disjointState. Each variable in the disjointState's varmap is mapped to a binding over the union's relvar by adding a binding to a relvarattr composed of the relvar name and the alias selected for that var in the subselect (e.g. union1.who). Constraints derived from earlier bindings are captured in the unionExpression. The unionExpression captures the assymmetric bindings, such as the fact that ?who is bound only in the second disjoint above. The disjointSubselect is composed of an attrlist, tableJoinList, and graphPatternExpression. The latter two come directly from the disjointState, and the attrlist is the disjoint number followed by a mapping from the unionVars to the the union relvar.variable name if the varmap maps that variable, or NULL otherwise (NULL AS bday above).

The SQL UNION of these disjointSubselects is joined as the relvar name, with a constraint of the unionExpression.

### 4.7 Optionals

SPARQL OPTIONALs are treated similarly to UNIONS; they are, in the general case, represented as a subselect, and an extra relvarattr is assigned to some non-NULL
constant in order to test if the LEFT OUTER JOIN on the subselect passed its ON constraint. Some OPTIONALs can be expressed as simple LEFT OUTER JOINs on a relvar (i.e., no subselect is required), but that is an optimization not discussed here.

Following is a query of employees (emp) and if they manage someone (flunky), that flunky’s name and department, and, if they have a manager, that manager’s name and department. The coreference between the first and second OPTIONAL demands that the flunky and the manager are in the same department.

```sparql
PREFIX emplP: <http://hr.example/DB/Employee#>
PREFIX mangP: <http://hr.example/DB/Manage#>
SELECT ?empName ?mangName ?flnkName ?dept
WHERE {
  ?emp emplP:lastName ?empName .
  OPTIONAL { ?flunky emplP:manager ?emp .
    ?flunky emplP:lastName ?flnkName .
    ?flunky emplP:department ?dept }
    ?manager emplP:lastName ?mangName .
    ?manager emplP:department ?dept }
}
```

While the following SQL could be simplified to eliminate LEFT OUTER JOINs, it is left here in a general form which maps the OPTIONAL to a LEFT OUTER JOIN of a subselect. The enforcement of the coreference on dept is described below in Equivalence of Optionally Bound Variables.

```sql
SELECT emp.lastName AS empName, opt1.mangName AS mangName, opt2.flnkName AS flnkName, IF(opt1._DISJOINT_ IS NOT NULL, opt1.dept, IF(opt2._DISJOINT_ IS NOT NULL, opt2.dept, NULL)) AS dept
FROM Employee AS emp
LEFT OUTER JOIN (
  SELECT 0 AS _DISJOINT_, manager.department AS dept,
  manager.empid AS manager, manager.lastName AS mangName
  FROM Employee AS manager
  WHERE manager.department IS NOT NULL
  AND manager.empid IS NOT NULL
  AND manager.lastName IS NOT NULL
) AS opt1 ON opt1.manager=emp.manager
LEFT OUTER JOIN (
  SELECT 0 AS _DISJOINT_, flunky.department AS dept,
  flunky.empid AS flunky,
  flunky.lastName AS flnkName, flunky.manager AS emp
  FROM Employee AS flunky
  WHERE flunky.department IS NOT NULL AND flunky.empid IS NOT NULL
  AND flunky.lastName IS NOT NULL AND flunky.manager IS NOT NULL
) AS opt2 ON opt2.emp=emp.empid
  AND (opt1._DISJOINT_ IS NULL OR opt2.dept=opt1.dept)
```
4.8 Equivalence of Optionally Bound Variables

SPARQL UNIONs with asymmetric bindings and SPARQL OPTIONALs have a semantics where a variable is conditionally bound, here called a "partial binding". A partial binding for a variable is a set of pairs of a binding constraint and a relvarattr. Our mapping to SQL has required that variables in SPARQL are bound to relvarattrs in SQL. Partially bound variables must be bound to an expression which will evaluate to the first binding of the variable to a non-NULL relvarattr, or to NULL if no relvarattr bound that variable. A partial coreference constraint for a new (potentially partial) binding is an SQL equivalence expression (a.b=c.d) between the new binding and all of the existing members of the set of partial bindings.

The first occurrence of ?dept in the SPARQL query demands that, if the first OPTIONAL is matched, ?dept is bound to the matched value. This is recorded in the varmap as by binding the variable to a set of tuples of the binding constraint and the relvarattr to which the variable is bound. After mapGraphPattern on the first OPTIONAL, ?dept is bound to \{ (opt1._DISJOINT_ IS NULL, opt1.dept) \}. The second occurrence demands that, if the first OPTIONAL was matched, any matching of the second OPTIONAL must have the same value for ?dept. The second optional binds ?dept to opt2.dept, so the partial coreference constraint against opt2.dept is the SQL expression (opt1._DISJOINT_ IS NULL OR opt2.dept=opt1.dept). A subsequent binding (not included in this example) to e.g. emp.dept would include both of the earlier constraints: (opt1._DISJOINT_ IS NULL OR emp.dept=opt1.dept) AND (opt2._DISJOINT_ IS NULL OR emp.dept=opt2.dept). Once a variable has been fully bound, it remains fully bound; all partial bindings are discarded.

Projecting partially bound variables into an attrlist requires that a solution which successfully binds a variable select that relvarattr. If the variable was never bound in the solution, we use the SQL term NULL to stand for a lack of binding. This can be implemented in conventional SQL by a nested if else expression which selects the relvarattr of first successful binding of the variable. Above, ?dept is bound to either opt1.dept, opt2.dept or NULL. If a variable becomes full bound through a reference to it in a non-optional graph pattern, the corresponding relvarattr in the attrlist will simply be that fully bound relvarattr.

4.9 Leading Optionals

Pushing the ?emp emplP:lastName ?empName constraint to the bottom is a very different, but illustrative query:

PREFIX emplP: <http://hr.example/DB/Employee#>
PREFIX mangP: <http://hr.example/DB/Manage#>
SELECT ?empName ?mangName ?flnkName ?dept
WHERE { OPTIONAL { ?flunky emplP:manager ?emp .
  ?flunky emplP:lastName ?flnkName .
  ?flunky emplP:department ?dept }
  ?emp emplP:lastName ?empName .
  prefixes { emplP: <http://hr.example/DB/Employee#> } }
    ?manager emplP:lastName ?mangName .
    ?manager emplP:department ?dept }.

The SPARQL semantics assert that WHERE { OPTIONAL { X } } is equivalent to WHERE { { } OPTIONAL { X } } and that the result set after processing { } contains one solution with no bindings. SQL has no syntax for starting off with an optional, but these semantics can be emulated by injecting into the tablelist a subselect with one (ignored) solution: ( SELECT 1 AS _EMPTY_ ) AS _EMPTY_. The selected value is unimportant, as is the relvar name so long as it does not conflict with another relvar name.

SELECT emp.lastName AS empName, opt1.grandManageName
FROM ( SELECT 1 AS _EMPTY_ ) AS _EMPTY_
LEFT OUTER JOIN ( 
    SELECT 0 AS _DISJOINT_, manager.department AS dept,
    manager.empid AS manager, manager.lastName AS mangName
    FROM Employee AS manager
    WHERE manager.department IS NOT NULL
    AND manager.empid IS NOT NULL
    AND manager.lastName IS NOT NULL
) AS opt1 ON TRUE
LEFT OUTER JOIN ( 
    SELECT 0 AS _DISJOINT_, flunky.department AS dept,
    flunky.empid AS flunky, flunky.lastName AS flnkName,
    flunky.manager AS emp
    FROM Employee AS flunky
    WHERE flunky.department IS NOT NULL
    AND flunky.empid IS NOT NULL
    AND flunky.lastName IS NOT NULL
    AND flunky.manager IS NOT NULL
) AS opt2 ON (opt1._DISJOINT_ IS NULL OR opt2.dept=opt1.dept)
INNER JOIN Employee AS emp
ON (opt1._DISJOINT_ IS NULL OR emp.manager=opt1.manager)
AND (opt2._DISJOINT_ IS NULL OR emp.empid=opt2.empid)

5 Related Work

RDF query languages have been used over triple stores for at least seven years. As SPARQL began to coalesce, Perez et al [SSEM] explored the emerging semantics of SPARQL and compared them to SQL. Cyganiak [SPALG] extended this semantics to include most of SPARQL, eliding some corner cases around variables introduced in nested optionals. Harris presented a system using hashes
for efficient triple storage and query in SQL [SHASH], including some practical datatype interoperability. Chebotko et al [SOPT] extended that to treat the translation of SPARQL OPTIONAL to SQL LEFT OUTER JOIN, including SQL’s three-state logic.

SPARQL-to-SQL [s2s] and Ultrawrap [ULT] create triple views in the relational store, where the view is a union of selects from the set of conventional relations to be exposed to SPARQL. These systems then map SPARQL to SQL queries over the triples relation, relying on the performance of mature SQL optimizer and execution engines.

Systems like D2R [D2R], FeDeRate [FDR], Virtuoso [VOS], Tripify [TRPL], SquirrelRDF [SQRL], DartQuery [DART] and Anzo [ANZO] addressed access to “legacy” data by extending this technology to include conventional (normalized) relational databases, but without providing a solid semantics upon which to base expectations and interoperability.

On the SQL side, the lack of a semantics in the specification led to the independent development of semantics, e.g. Negri et al, Formal semantics of SQL queries [SSQL]. Projects like SchemaSQL [SSQL] attempted to extend SQL to gracefully address addressing across different databases. While SSQL proposed a query language with variables in the graph pattern (like SPARQL), it did not introduce the global identifiers and binary propositions which characterize RDF.

Prolog’s propositional structure is similar to RDF, and SPARQL BGP matching can be implemented using prolog unification. P2R [P2R] and TREQL [TREQL] are query languages which map prolog unification to SQL queries. The effort by Maier et al to develop a similar system, NED-2 [NED], which does not require a priori knowledge of the relational schema, could inspire a more agile variant of the algebra presented in this paper.

6 Conclusion

The algebra described in this document was implemented and tested in a strongly-typed Scala implementation [SCALA] in less than two weeks. Extending this to other implementations will allow database vendors and data custodians to create a fabric of inter-linked data, allowing users much more agility in using this data in ways they never could before.

Earlier implementations of this algebra in SPASQL [SPASQL] and FeDeRate [FDR] demonstrated the practicality of this algebra. The message for all of this work has been that using the Semantic Web introduces no query execution performance penalty over conventional relational databases.

References

A Mapping of SPARQL Onto Conventional SQL


