Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns

(Technical Report TR-DB-052006-CLJF, May 2006. Revised November 2006)

Artem Chebotko, Shiyong Lu, Hasan M. Jamil and Farshad Fotouhi
Wayne State University
Department of Computer Science
5143 Cass Avenue, Detroit, Michigan 48202, USA
artem@cs.wayne.edu

Abstract

The Semantic Web has recently gained tremendous momentum due to its great potential for providing a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. While Resource Description Framework (RDF) is a W3C recommended langauge for representing data over the Semantic Web, SPARQL is an emerging W3C query language for RDF data. Although several researchers have proposed to use RDBMSs to store and query RDF data, to our best knowledge, there is no published algorithm for translating a SPARQL query to an equivalent SQL counterpart in the presence of arbitrary complex optional graph patterns. In this paper, we propose: (i) a basic graph pattern translation algorithm, BGPtoSQL, that translates a basic graph pattern to its SQL equivalent; and based on BGPtoSQL, (ii) a semantics preserving SPARQL-to-SQL query translation algorithm, SPARQLtoSQL, for SPARQL queries that contain arbitrary complex optional graph patterns. Experimental results show that our algorithms are efficient and scalable.

1 Introduction

The Semantic Web has recently gained tremendous momentum due to its great potential for providing a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. While Resource Description Framework (RDF) [5] is the standard language for annotating resources on the Semantic Web [14], RDF Schema [6] is the standard language for specifying a vocabulary of terms for a particular application domain for the purpose of annotation. RDF data consists of a set of statements, called *triples*, of the form `<s,p,o>`, where `s` is a subject, `p` is a predicate and `o` is an object. The set of triples can be represented as a directed graph, in which nodes represent subjects and objects, and edges represent predicates connecting from subject nodes to object nodes.

SPARQL [10] is a query language for RDF that has been recently proposed by the World Wide Web Consortium. SPARQL allows the specification of triple and graph patterns to be matched over RDF graphs. Example 1.1 presents a sample SPARQL query with both parallel and nested `OPTIONAL` graph patterns, which returns the name of a person, the country of birth, and (1) a social security number if it is available, or (2) a passport number if it is available, and those countries which have issued visas to the passport if there are any such countries.

Example 1.1 (SPARQL query)

```
01     SELECT ?name ?birthcountry ?number ?country
02     WHERE {
03       ?someone rdf:type :Person .
04       ?someone :name ?name .
05       ?someone :birthcountry ?birthcountry .
06       OPTIONAL {?someone :ssn ?number}
```

```
07       OPTIONAL {
08         ?someone :passportno ?number .
09         OPTIONAL { ?number :visacountry ?country }
10       }
11     }
```

The `WHERE` clause in this example contains both non-optional and optional parts. The non-optional part is the basic graph pattern defined with three triple patterns in lines 03-05. The basic graph pattern searches for the instances of class `Person` which have a name and a country of birth. The non-optional part must match for the query to succeed. Therefore, variables `?someone`, `?name`, and `?birthcountry` must be bound.

The optional part includes three `OPTIONAL` clauses and does not have to match for the query to succeed. The first `OPTIONAL` clause in line 06 searches for a person's social security number. The second `OPTIONAL` clause in line 07 searches for a person's passport number, and if the number is matched, then the third `OPTIONAL` clause in line 09 will further search for those countries which have issued visas to the passport. For a given value of `?someone`, if `?number` has already been bound to some value in line 06, then `?number` must be bound to the same value in line 08. If `?number` is unbound in line 06, then `?number` can be bound to any value in line 08. For the third `OPTIONAL` clause to augment the query result, its triple pattern in line 09 must match and the second `OPTIONAL` clause must have succeeded. ♢

The above example illustrates several challenges of SPARQL query processing with the presence of `OPTIONAL` patterns with the complexity of the semantics of shared variables, parallel `OPTIONAL` clauses, and nested `OPTIONAL` clauses.

- *Basic semantics of `OPTIONAL` patterns.* The evaluation of an `OPTIONAL` clause is not obligated to succeed, and in the case of failure, no value will be returned for those unbound variables in the `SELECT` clause. In this work, for convenience of the presentation, we denote the absence of a value by a `NULL` value.
- *Semantics of shared variables.* In general, shared variables must be bound to the same values. Variables can be shared among subjects, predicates, objects, and across each other.
- *Semantics of parallel `OPTIONAL` patterns.* While the failure of the evaluation of an `OPTIONAL` clause does not block the evaluation of a following parallel `OPTIONAL` clause, the success of the evaluation of an `OPTIONAL` clause obligates the same variables in the following parallel `OPTIONAL` clauses to be bound to the same values.
- *Semantics of nested `OPTIONAL` patterns.* Before an `OPTIONAL` clause is evaluated, all containing basic graph patterns or `OPTIONAL` clauses must have succeeded.

Although numerous researchers [18, 16, 26, 25, 19, 30, 15, 29, 22] have proposed to use RDBMSs to store and query RDF data using the SQL and SPARQL query languages, to the best our knowledge, there is no published algorithm for translating a SPARQL query to an equivalent SQL counterpart in the presence of arbitrary complex optional graph patterns. Although Cyganiak defines a relational algebra for SPARQL and outlines a set of rules to establish the equivalence between this algebra and SQL, the nested `OPTIONAL` pattern problem still lacks a full solution as noted by the author, "Unfortunately, the join rule stated above does not fully reproduce SPARQL semantics" [18]. In the meanwhile, Harris et al. [19] present an approach for translating SPARQL queries into SQL; again, the authors did not give a solution to the nested `OPTIONAL` pattern problem as he noted "A more sophisticated algorithm is required to express nested optional graph patterns" [19]. Finally, it is worthwhile to mention a few RDBMS based SPARQL query processing tools, including ARQ [2], Virtuoso Universal Server [27], sparql2sql [11], Sesame 2 [9], KAON2 [3], 3store [19], and ARC SPARQL2SQL Rewriter [1], although most of them are still under active development and their implementation details are not clear.

The main contributions of this paper are:

- We propose a basic graph pattern translation algorithm, BGPtoSQL, that translates a basic graph pattern to its SQL equivalent.[1]

[1] We believe that existing RDF toolkits employ similar algorithms for the basic graph pattern translation.

- Based on BGPtoSQL, we propose a semantics preserving SPARQL-to-SQL query translation algorithm for SPARQL queries that contain arbitrary complex optional graph patterns. To our best knowledge, this is the first full algorithm for translating a SPARQL query to an equivalent SQL counterpart in the presence of arbitrary complex optional graph patterns, including both parallel and nested graph patterns.
- We conduct a number of experiments to show that our algorithms are efficient and scalable.

Organization. The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 presents the preliminaries for the paper, including an overview of used relational RDF storage and the notations used in the paper. Section 4 presents the BGPtoSQL algorithm that generates an SQL query equivalent to an input SPARQL basic graph pattern. Section 5 describes our SPARQL-to-SQL translation strategy. Section 6 presents the SPARQLtoSQL algorithm that translates an input SPARQL query with a basic graph pattern or an optional graph pattern to an equivalent SQL query. Section 7 reports the results of the experimental study. Finally, we provide our conclusions and future work in Section 8.

2 Related Work

A number of RDF query languages have been proposed, including SPARQL [10], RDQL [8], RQL [21], SeRQL [12], and RDFQL [7]. These languages are declarative in nature and bear an SQL-style syntax. Among them, SPARQL, SeRQL, and RDFQL support optional graph patterns.

Numerous researchers [18, 16, 26, 25, 19, 30, 15, 29, 22] have proposed to use RDBMSs to store and query RDF data using the SQL and SPARQL query languages. One of the most challenging problems in such an approach is the translation of SPARQL queries into relational algebra and SQL. Most of the existing papers focus on the translation of basic graph patterns using inner joins and "flat" optional graph patterns using left outer joins and only a few consider the SPARQL-to-SQL translation in the presence of nested `OPTIONAL`s. Although Cyganiak defines a relational algebra for SPARQL and outlines a set of rules to establish the equivalence between this algebra and SQL, the nested `OPTIONAL` pattern problem still lacks a full solution as noted by the author, "Unfortunately, the join rule stated above does not fully reproduce SPARQL semantics" [18]. In the meanwhile, Harris et al. [19] presents an approach for translating SPARQL queries into SQL; again, the authors did not give a solution to the nested `OPTIONAL` pattern problem as he noted "A more sophisticated algorithm is required to express nested optional graph patterns" [19]. To our best knowledge, there is no published algorithm for translating a SPARQL query to an equivalent SQL counterpart in the presence of arbitrary complex optional graph patterns.

Perez et al. [24, 23] advocate the compositional semantics of SPARQL which has some differences with respect to the last official SPARQL release [10] (4th October 2006). In the context of optional graph patterns, the major advantage of the compositional semantics is that it allows simpler SPARQL-to-SQL translation for optional patterns (e.g., the `NOT NULL` check is not required) and enables better optimization (e.g., the order of the evaluation of `OPTIONAL` clauses can be relaxed based on the equivalences in [24]).

It is worthwhile to mention the following SPARQL query processing tools: ARQ [2], Virtuoso Universal Server [27], sparql2sql [11], Sesame 2 [9], KAON2 [3], 3store [19], Rasqal [4], and ARC SPARQL2SQL Rewriter [1]. However, most of these systems are still under active development and their implementation details are not clear.

Finally, a couple of researchers proposed to extend SQL or RDQL to support the query of RDF data. Chong et al. [17] introduce an SQL table function, `RDF_MATCH`, to query RDF data, such that the function can be combined with SQL statements for further processing. One of the input parameters of `RDF_MATCH` is a graph pattern which semantically corresponds to a basic graph pattern defined in SPARQL. Hung et al. [20] study the problem of RDF aggregate queries by extending the RDQL query language with the `GROUP BY` clause and several aggregate functions (e.g., *max* and *count*).

3 Preliminaries

In this paper, we use a single table `Triples(subject,predicate,object)` to store RDF triples, such that each triple is naturally mapped to one row of the table. This triple store scheme, although not as efficient as

some other storage schemas [28], is the best for our presentation purposes due to its simplicity and application independence. In the following, we define the basic graph pattern model and SPARQL query model as data structures for our algorithms.

Definition 3.1 (Basic Graph Pattern Model) A basic graph pattern is modeled as a directed graph $\mathcal{BGP} = (N, E)$, where N is a set of nodes representing subjects and objects, and E is a set of edges representing predicates. Each edge is directed from a subject node to an object node. Each node is labeled (attribute *label*) with a variable name, a URI, a blank node, or a literal, and each edge is labeled with a variable name or a URI. Additionally, each node has pointers to all incoming and outgoing edges and their quantities in the *in-degree* and *out-degree* attributes, respectively. ♢

The SPARQL query in Example 1.1 has four basic graph patterns: the first basic graph pattern has three triple patterns (lines 03-05), the second basic graph pattern has one triple pattern (line 06) and so forth. Their basic graph pattern models are visualized in Figure 1.

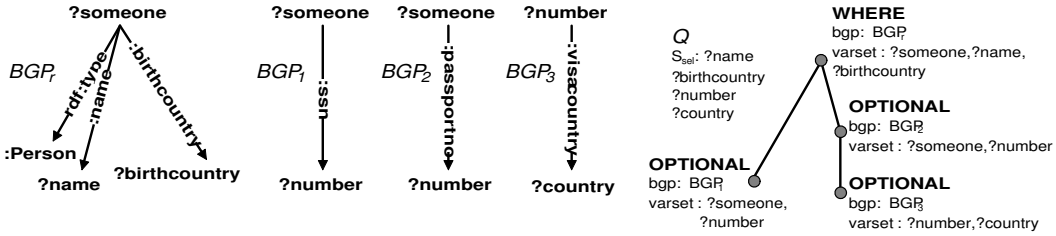

Figure 1: Basic graph patterns and a SPARQL query model for Example 1.1

Definition 3.2 (SPARQL Query Model) A SPARQL query $\mathcal{Q}$ with a basic graph pattern or an optional graph pattern in its `WHERE` clause is modeled as a tuple (S_{sel}, T), where S_{sel} is the ordered list of variables in the `SELECT` clause and $T = (N, E, r, bgp, varset)$ is the ordered tree of clauses rooted at the `WHERE` clause of $\mathcal{Q}$, where N is a set of nodes representing clauses in $\mathcal{Q}$; E is a set of edges representing the containment relationship between two clauses, such that there exists an edge between nodes n_1 and n_2 if and only if the clause of n_2 is directly contained (nested) in the clause of n_1; r is the root of the tree and corresponds to the `WHERE` clause; *bgp* is a function that annotates each node with a basic graph pattern that is constructed from triple patterns directly contained in the corresponding clause, but not in nested clauses; and $varset$ is a function that annotates each node with the set of variables directly found in the corresponding clause, but not in nested clauses. Finally, T is ordered such that the clause precedence in $\mathcal{Q}$ is preserved for corresponding nodes. ♢

The SPARQL query model of Example 1.1 is illustrated in Figure 1.

4 Generating SQL Queries for Basic Graph Patterns

Algorithm BGPtoSQL is a primitive for translating a basic graph pattern into an equivalent SQL query, such that the SQL query retrieves RDF subgraphs matching the graph pattern from the triple store. The SQL query result is a relation whose schema is the set of variables found in the graph pattern and whose every tuple is the set of variable bindings that correspond to a particular RDF subgraph matched by the graph pattern.

BGPtoSQL is given in Figure 2. The algorithm treats blank nodes as a special case of a variable with the scope of a basic graph pattern. Therefore, the algorithm substitutes every blank node label in the input graph pattern BGP with a unique variable in line 05, such that multiple occurrences of the same blank node are substituted by the same variable. The uniqueness property should hold for the scope of a SPARQL query to ensure that blank nodes in one basic graph pattern will not coincide with blank nodes in another pattern.

The algorithm constructs the `FROM` and `SELECT` clauses of the to-be-returned SQL query in lines 06-07 and lines 08-19, respectively. For each edge in BGP, a unique alias of table `Triples` is generated, and all the aliases are added to the `FROM` clause. All distinct variables in BGP are projected in the `SELECT` clause, such that a predicate/subject/object variable is represented by the corresponding column of the `Triples` table.

```
01 Algorithm BGPtoSQL
02 Input: basic graph pattern BGP = (N, E)
03 Output: SQL query
04 Begin
05 Substitute each distinct blank node label in BGP with a unique variable /* unique for the scope of a SPARQL query */
06 Assign each edge e ∈ E a unique table alias t_e /*unique for the scope of a SPARQL query*/
07 Construct the FROM clause to contain all the table aliases /*from += "Triples $t_e" for each e ∈ E*/
08     For each distinct variable v in BGP do /* Construct the SELECT clause */
09         If v is a predicate variable then
10             Let e be the corresponding edge
11             select += "$t_e.predicate AS $e.label"
12         ElseIf v is an object variable then
13             Let e be the first incoming edge of the corresponding node n
14             select += "$t_e.object AS $n.label"
15         Else /* v is a subject variable */
16             Let e be the first outgoing edge of the corresponding node n
17             select += "$t_e.subject AS $n.label"
18         End If
19     End For
20     For each RDF term (a URI or a literal) m in BGP do /* Construct the WHERE clause */
21         If m is a predicate term then
22             Let e be the corresponding edge
23             where += "$t_e.predicate = $e.label AND "
24         Else /* m is an object or a subject term */
25             Let n be the corresponding node
26             For each incoming edge e of n do
27                 where += "$t_e.object = $n.label AND " End For
28             For each outgoing edge e of n do
29                 where += "$t_e.subject = $n.label AND " End For
30         End If
31     End For
32     For each node n ∈ N labeled with a variable do
33         If n.in-degree > 1 then /* Case 1 */
34             Let e_1^in be the first incoming edge of n
35             For each incoming edge e_i^in of n and e_i^in ≠ e_1^in do
36                 where += "$t_{e_1^in}.object = $t_{e_i^in}.object AND " End For
37         End If
38         If n.out-degree > 1 then /* Case 2 */
39             Let e_1^out be the first outgoing edge of n
40             For each outgoing edge e_i^out of n and e_i^out ≠ e_1^out do
41                 where += "$t_{e_1^out}.subject = $t_{e_i^out}.subject AND " End For
42         End If
43         If n.in-degree > 0 && n.out-degree > 0 then /* Case 3 */
44             Let e_1^in be the first incoming edge of n; Let e_1^out be the first outgoing edge of n
45             where += "$t_{e_1^in}.object = $t_{e_1^out}.subject AND "
46         End If
47     End For
48     For each distinct predicate variable and the corresponding edge e ∈ E do
49         For each edge e_i ∈ E && e_i ≠ e && e_i.label = e.label do /* Case 4 */
50             where += "$t_e.predicate = $t_{e_i}.predicate AND " End For
51         For each node n_i ∈ N && n_i.label = e.label do /* Case 5 */
52             Let e_1 be the first incoming/outgoing edge of n_i /* '/' means 'or' */
53             where += "$t_e.predicate = $t_{e_1}.object/subject AND " End For
54     End For
55     Return "SELECT" + select + "FROM" + from + "WHERE" + where
56 End Algorithm
```

Figure 2: Algorithm BGPtoSQL

The `WHERE` clause is constructed in lines 20-54. First, in lines 20-31, all restrictions with respect to labeling (e.g., the *label* attribute is a URI or a literal) are included. Second, shared variables require unique instantiation and thus must participate in join conditions. There are five cases (see Figure 3) that require self joins of the `Triples` table:

Case 1 If node n is labeled with a variable and has multiple incoming edges, then the `object` attributes of

all table aliases that correspond to these incoming edges must be equal (lines 33-37).

Case 2 If node n is labeled with a variable and has multiple outgoing edges, then the `subject` attributes of all table aliases that correspond to these outgoing edges must be equal (lines 38-42).

Case 3 If node n is labeled with a variable and has an incoming edge and an outgoing edge, then the `object` attribute of the incoming edge table alias must be equal to the `subject` attribute of the outgoing edge table alias (lines 43-46).

Case 4 If edge e is labeled with a variable and there exists another edge that is labeled with the same variable, then the `predicate` attributes of the table aliases that correspond to the two edges must be equal (lines 49-50).

Case 5 If edge e is labeled with a variable and there exists a node labeled with the same variable, then the `predicate` attribute of the edge table alias must be equal to the `object` (`subject`) attribute of the node's incoming (outgoing) table alias (lines 51-53).

Finally, the constructed SQL query is returned in line 55.

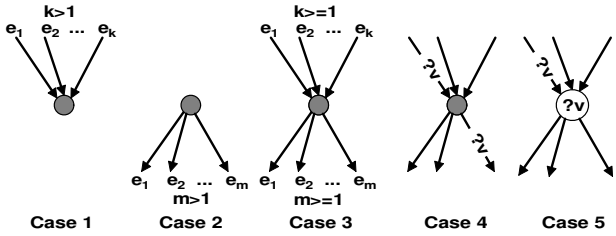

Figure 3: Five cases that require self joins of the Triples table

Example 4.1 (BGPtoSQL) Consider the following basic graph pattern (its model is illustrated in Figure 4, where t1, t2, t3 and t4 are the `Triples` table aliases generated for the corresponding edges) and the SQL statement generated by BGPtoSQL. Since the graph pattern has four triples (edges), the query has four tables in the `FROM` clause. The URI terms are translated as shown in lines 04-05. The join conditions are shown in lines 06-08. The `SELECT` clause projects the three variables that appear in the pattern.

```
01  ?a :p1 :?b . ?a :p2 :uri . ?b ?a ?c . :uri ?a ?c .
02  SELECT t3.predicate AS a, t1.object AS b, t3.object AS c
03  FROM   Triples t1, Triples t2, Triples t3, Triples t4
04  WHERE  t1.predicate = ':p1' AND t2.predicate = ':p2'
05    AND  t2.object = ':uri' AND t4.subject = ':uri'
06    AND  t3.object = t4.object /*Case 1*/ AND t1.subject = t2.subject /*Case 2*/
07    AND  t1.object = t3.subject /*Case 3*/ AND t3.predicate = t4.predicate /*Case 4*/
08    AND  t3.predicate = t1.subject /*Case 5*/
```

♢

5 SPARQL-to-SQL Query Translation Strategy

Given a SPARQL query $\mathcal{Q} = (S_{sel}, T)$ and $T = (N, E, r, bgp, varset)$, in this section, we show by examples how our strategy for SPARQL-to-SQL query translation works.

First, for each $\mathcal{BGP}_i \in \{bgp(n_i) | n_i \in N\}$, we use BGPtoSQL primitive to generate an equivalent SQL query and store the result in relation R_i with attributes $varset(n_i)$. Our goal is to join these relations into one relation R_{res} under the SPARQL semantics.

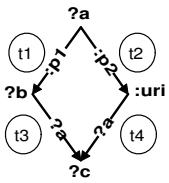

Figure 4: Basic graph pattern model for Example 4.1

Second, when joining two relations, we use left outer join ($\ltimes$) to preserve all tuples of the first relation in the resulting relation as shown in Example 5.1. The left outer join captures the basic semantics of `OPTIONAL` patterns: the evaluation of an `OPTIONAL` clause is not obligated to succeed, and in the case of failure, a `NULL` value will be returned for unbound variables.

Example 5.1 (Left outer join)

```
01     :x :p1 "1"; :p1 "11".
02     SELECT ?b ?c
03     WHERE {
04       ?a :p1 ?b .
05       OPTIONAL { ?a :p2 ?c }
06     }
```

Consider the RDF data and the SPARQL query given above. Let $\mathcal{BGP}_r$ and $\mathcal{BGP}_1$ are constructed from the corresponding graph patterns in lines 04 and 05. We have $R_r(a, b) = \{(: x, 1), (: x, 11)\}$ and $R_1(a, c) = \{\}$ as their corresponding relations. The resulting relation R_{res} is computed as:

$R_{res} = \Pi_{R_r.b,R_1.c}(R_r \ltimes_{R_r.a=R_1.a} R_1)$,
$R_{res}(b, c) = \{(1, \texttt{NULL}), (11, \texttt{NULL})\}$.

The inner join will fail in this case, resulting in an empty relation. ♢

Third, to preserve the left-associativity semantics of `OPTIONAL` clauses, our strategy employs the preorder traversal of tree T to join the relations, such that each visited edge corresponds to a left outer join. For example, the first join is always between the relation R_r that corresponds to the root r of T and the relation that corresponds to r's first child. The result of this join is assigned to the relation denoted as R_{res}. Next, R_{res} is joined with r's first grandchild if any, otherwise, with r's second child. Again, the result is assigned to R_{res}. After each join, all distinct relational attributes are projected, since they may participate in following join conditions. After the last join, the attributes that correspond to the variables in the SPARQL `SELECT` clause are projected. The following example illustrates the importance of the join order.

Example 5.2 ((Join order))

```
01     :x :p1 "1"; :p1 "11"; :p2 :y; :p3 :z.
02     SELECT ?b ?c
03     WHERE {
04       ?a :p1 ?b .
05       OPTIONAL {
06         ?a :p2 ?c .
07         OPTIONAL { ?a :p3 ?c }
08       }
09     }
```

Consider the RDF data and the SPARQL query given above. Let $\mathcal{BGP}_r$, $\mathcal{BGP}_1$ and $\mathcal{BGP}_2$ are constructed from the corresponding graph patterns in lines 04, 06 and 07. We have $R_r(a, b) = \{(: x, 1), (: x, 11)\}$, $R_1(a, c) = \{(: x, : y)\}$ and $R_2(a, c) = \{(: x, : z)\}$ as their corresponding relations. We compute the resulting relation in the following manner:

$R_{res} = \Pi_{R_r.a,R_r.b,R_1.c}(R_r \ltimes_{R_r.a=R_1.a} R_1)$,
$R_{res} = \Pi_{R_{res}.b,R_{res}.c}(R_{res} \ltimes_{R_{res}.a=R_2.a \wedge R_{res}.c=R_2.c} R_2)$,
$R_{res}(b,c) = \{(1,:y),(11,:y)\}$

If we choose to join R_r and R_2 first, the result will be incorrect. ♢

Finally, when joining relations R_{res}/R_r and R_i, their common attributes are added to the join condition to be equal, however additional conditions may apply: (1) if R_i corresponds to a nested `OPTIONAL` clause, then for the join to succeed, its parent clause must have succeeded and (2) if R_i corresponds to a parallel `OPTIONAL` clause and there is a preceding parallel `OPTIONAL` clause that has not succeeded, then the attributes/variables that are shared by the parallel clauses but not by their ancestor clauses do not have to be equal. These conditions capture the semantics of nested and parallel `OPTIONAL` clauses and are enforced by our translation as shown in the following examples.

Example 5.3 (Nested `OPTIONAL` clauses and unbound variables)

```
01     :x :p1 :y; :p1 :z; :p3 "3".
02     SELECT ?b ?c ?d
03     WHERE {
04       ?a :p1 ?b .
05       OPTIONAL {
06         ?a :p2 ?c .
07         OPTIONAL { ?b :p3 ?d }
08       }
09     }
```

Consider the RDF data and the SPARQL query given above. Let $\mathcal{BGP}_r$, $\mathcal{BGP}_1$ and $\mathcal{BGP}_2$ are constructed from the corresponding graph patterns in lines 04, 06 and 07. We have $R_r(a,b) = \{(:x,:y),(:x,:z)\}$, $R_1(a,c) = \{\}$ and $R_2(b,d) = \{(:x,3)\}$ as their corresponding relations. We compute the resulting relation in the following manner:

$R_{res} = \Pi_{R_r.a,R_r.b,R_1.c}(R_r \ltimes_{R_r.a=R_1.a} R_1)$,
$R_{res} = \Pi_{R_{res}.b,R_{res}.c,R_2.d}(R_{res} \ltimes_{R_{res}.b=R_2.b \wedge R_{res}.c \texttt{ IS NOT NULL}} R_2)$,
$R_{res}(b,c,d) = \{(:y,\texttt{NULL},\texttt{NULL}),(:z,\texttt{NULL},\texttt{NULL})\}$

The condition "$R_{res}.c$ `IS NOT NULL`" is introduced to ensure that `?c` has been bound before the `OPTIONAL` clause in line 07 is evaluated. `?c` is used as an indicator whether the containing graph pattern in line 06 has succeeded or not. ♢

Example 5.4 (Parallel `OPTIONAL` clauses and unbound variables)

```
01     :x :p1 "1"; :p2 :y; :p3 :y. :y :p4 "4". :z :p1 "11"; :p3 :y.
02     SELECT ?b ?c ?d
03     WHERE {
04       ?a :p1 ?b .
05       OPTIONAL { ?a :p2 ?c } .
06       OPTIONAL { ?a :p3 ?c . ?c :p4 ?d }
07     }
```

Consider the RDF data and the SPARQL query given above. Let $\mathcal{BGP}_r$, $\mathcal{BGP}_1$ and $\mathcal{BGP}_2$ are constructed from the corresponding graph patterns in lines 04, 05 and 06. We have $R_r(a,b) = \{(:x,1),(:z,11)\}$, $R_1(a,c) = \{(:x,:y)\}$ and $R_2(a,c,d) = \{(:x,:y,4),(:z,:y,4)\}$ are their corresponding relations. We compute the resulting relation in the following manner:

$R_{res} = \Pi_{R_r.a,R_r.b,R_1.c}(R_r \ltimes_{R_r.a=R_1.a} R_1)$, /*$R_{res}(a,b,c) = \{(:x,1,:y),(:z,11,\texttt{NULL})\}$*/
$R_{res} = \Pi_{R_{res}.b,\texttt{COALESCE}(R_{res}.c,R_2.c),R_2.d}(R_{res} \ltimes_{R_{res}.a=R_2.a \wedge (R_{res}.c=R_2.c \vee R_{res}.c \texttt{ IS NULL})} R_2)$,
$R_{res}(b,c,d) = \{(1,:y,4),(11,:y,4)\}$.

The condition "$R_{res}.c = R_2.c \vee R_{res}.c$ `IS NULL`" is introduced to ensure that (1) if `?c` is bound in line 05, it must be bound to the same value in the parallel clause in line 06 and (2) if `?c` is unbound in line 05, it can be bound to any value in line 06. Since `?c` can be bound in $\mathcal{BGP}_1$, $\mathcal{BGP}_2$ or both (to the same value), we use the SQL `COALESCE` function to return the binding of `?c`. The `COALESCE` function returns the first non-`NULL` value among its arguments or `NULL` if all arguments are `NULL` values. ♢

6 SPARQL-to-SQL Query Translation Algorithm

We present the SPARQLtoSQL algorithm that takes a SPARQL query with a basic graph pattern or an optional graph pattern and translates it into an equivalent SQL query that matches a graph pattern in a relational RDF storage. The algorithm uses the BGPtoSQL primitive for translating a basic graph pattern into an equivalent SQL query. If such a primitive is provided, SPARQLtoSQL can generate SQL queries regardless of the RDF storage scheme.

The SPARQLtoSQL algorithm is given in Figure 5. At the preprocessing stage, the algorithm substitutes every blank node label in each basic graph pattern of a query with a unique variable (line 06), such that multiple occurrences of the same blank node within one basic graph pattern are substituted by the same variable. The uniqueness property should hold for the scope of a query. Next, a unique variable is introduced into a basic graph pattern (lines 07-09), if (1) the pattern has no variables or (2) the pattern has no variables that are different from its ancestor or predecessor variables, and the graph pattern is in the `OPTIONAL` clause that contains a nested `OPTIONAL` clause. The introduced variable is to indicate if the clause pattern matches or not. The first condition is important for BGPtoSQL to generate a non-empty relation that serves as a result of a basic graph pattern match. The second condition is required to ensure that a containing `OPTIONAL` clause graph pattern has succeeded before its nested `OPTIONAL` clause is evaluated. For instance, when evaluating the third `OPTIONAL` clause (line 09) in Example 1.1, we must ensure that its parent clause (line 08) has succeeded by enforcing either `?someone IS NOT NULL` or `?number IS NOT NULL` in the join condition. However, both variables could have been bound before line 08, and therefore, the `NOT NULL` check may succeed even if the parent pattern has failed. Our solution is to introduce a unique variable in line 08. A particular implementation may substitute an RDF term (a URI or a literal) in a pattern with a variable and restrict the variable value to be equal to the term value in the SQL `WHERE` clause. In the case that a basic graph pattern contains only shared variables and no RDF terms, one can introduce a dummy triple pattern that always matches and has at least one variable and all its variables are unique.

The algorithm uses the stack-based preorder traversal of the query clause tree T to visit each node (clause) n. Every node n is annotated with a basic graph pattern which is translated into SQL using the BGPtoSQL algorithm, and the SQL code is assigned to q_n (line 19). For each next iteration, root r of T is visited, and q_n is assigned to Q_{sql}. At every next iteration, Q_{sql} is assigned the left outer join of Q_{sql} and q_n. Therefore, Q_{sql} contains an SQL query generated from graph patterns that correspond to the visited nodes of T. The new value for Q_{sql} is constructed as described in the following (lines 23-32).

Unique table aliases r_1 and r_2 are generated for Q_{sql} and q_n, respectively, and the left outer join of r_1 and r_2 is added to the `FROM` clause (lines 27-28). The join condition includes:

- Equality of shared columns/variables that are in both n and n's ancestors, since the variables must be bound to the same values (line 29). See Example 5.1 and variable `?a` for reference.
- `NOT NULL` check on one of the columns/variables of n's parent, such that the chosen column does not occur in ancestors or predecessors of n's parent (line 30). If n's parent is the root of T, the check is not required, since all the variables of root's basic graph pattern are always bound in the result. See Example 5.3 and variable `?c` for reference.
- Equality of shared columns/variables that are in both n and n's predecessors, excluding ancestors. However, if the variables are unbound in the predecessors, the equality must be logically "eliminated", since the variables can be bound in r_1 only, in r_2 only or in both r_1 and r_2 to the same values (line 31). See Example 5.4 and variable `?c` for reference.

The `SELECT` clause of Q_{sql} is constructed in lines 24-26 to project:

01 **Algorithm SPARQLtoSQL**
02 **Input:** SPARQL query Q = (S_{sel}, T) and $T = (N, E, r, bgp, varset)$
03 **Output:** SQL query
04 **Begin**
05 Preprocessing: /* hereafter, uniqueness is over the scope of Q */
06 In each $\mathcal{BGP}_i \in \{bgp(n_i) | n_i \in N\}$, substitute each distinct blank node label with a unique variable
07 Introduce a unique variable in each $\mathcal{BGP}_i \in \{bgp(n_i) | n_i \in N \wedge varset(n_i) = \emptyset\}$
08 Introduce a unique variable in each `OPTIONAL` pattern $\mathcal{BGP}_i \in \{bgp(n_i) | n_i \in N \wedge n_i \neq r\}$,
09 if n_i has a child and all the variables in $varset(n_i)$ also occur in n_i's ancestors or predecessors
10 Stack ST = EmptyStack()
11 ST.push(r)
12 **While** ST.isNotEmpty() **do**
13 n = ST.pop()
14 $S_n = varset(n)$ /* Let S_n be a set of variables found in n */
15 Let S_{n-anc} be a set of variables found in n's ancestors
16 Let S_{n-pre} be a set of variables found in n's ancestors and predecessors
17 Let S_p be a set of variables found in n's parent
18 Let S_{p-pre} be a set of variables found in n's parent ancestors and predecessors
19 q_n = BGPtoSQL($bgp(n)$)
20 **If** $n = r$ **then** $Q_{sql} = q_n$
21 **Else**
22 Generate two unique table aliases r_1 and r_2
23 Q_{sql} = "
24 `SELECT` $r_1.col$ `AS` $col \ \forall \ col \in S_{n-anc} \cup (S_{n-pre} - S_n)$,
25 $r_2.col$ `AS` $col \ \forall \ col \in S_n - S_{n-pre}$,
26 `COALESCE` ($r_1.col$, $r_2.col$) `AS` $col \ \forall \ col \in S_n \cap S_{n-pre} - S_{n-anc}$
27 `FROM` (Q_{sql}) r_1
28 `LEFT OUTER JOIN` (q_n) r_2
29 `ON` (($r_1.col = r_2.col \ \forall \ col \in S_n \cap S_{n-anc}$)
30 `AND` ($r_1.col$ `IS NOT NULL` for one $col \in S_p - S_{p-pre}$, if n's parent $\neq r$)
31 `AND` ($r_1.col = r_2.col$ `OR` $r_1.col$ `IS NULL` $\forall \ col \in S_n \cap S_{n-pre} - S_{n-anc}$)
32)"
33 **End If**
34 **For** c = n.lastChild(); $c \neq$ null; c = n.prevChild() **do** ST.push(c) **End For**
35 **End While**
36 Q_{sql} = "
37 `SELECT` $r.col$ `AS` $col \ \forall \ col \in S_{sel}$
38 `FROM` (Q_{sql}) `AS` r"
39 **Return** Q_{sql}
40 **End Algorithm**

Figure 5: Algorithm SPARQLtoSQL

- Columns/variables of r_1 that are (1) in n's ancestors or (2) in n's predecessors, but not in n. The variables of the first type, if bound in r_1, can be bound in r_2 only to the same values as in r_1. The variables of the second type are not in r_2, since they do not occur in n's basic graph pattern. See Example 5.2 and variables `?a` and `?b` for reference.

- Columns/variables of r_2 that are not in r_1. These variables are in n, but not in n's predecessors and ancestors. See Example 5.3 and variable `?c` for reference.

- Columns/variables that are in both n and n's predecessors, excluding n's ancestors. These variables can be bound in r_1 only, in r_2 only or in both r_1 and r_2 to the same values. Therefore, the SQL `COALESCE` function is used to return available bindings or `NULL` if variables are unbound in both r_1 and r_2. See Example 5.4 and variable `?c` for reference.

Variable sets that satisfy stated conditions for join and projection are computed using set operations. The initial sets used in the algorithm include S_n (variables in n), S_{n-anc} (variables in n's ancestors), S_{n-pre} (variables in n's predecessors), S_p (variables in n's parent) and S_{p-pre} (variables in n's parent predecessors). Some of them are schematically shown in Figure 6. Also, note that $S_1 = S_{n-anc} \cup (S_{n-pre} - S_n)$ (line 24), $S_2 = S_n - S_{n-pre}$ (line 25) and $S_3 = S_n \cap S_{n-pre} - S_{n-anc}$ (line 26) are mutually disjoint sets, and $S_1 \cup S_2 \cup S_3 = S_{n-pre} \cup S_n$. Therefore, all distinct columns are projected from r_1 and r_2.

Finally, once all nodes in T are visited, the columns that correspond to the variables in S_{sel} are projected from Q_{sql} (lines 36-38) and the resulting query is returned (line 39).

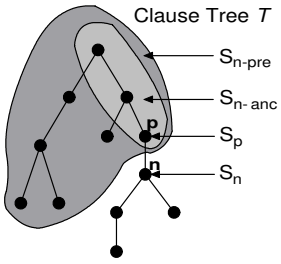

Figure 6: Variable sets in the clause tree of a query

7 Experimental Study

In this section, we present the performance evaluation of the proposed algorithms and SPARQL queries translated into SQL and issued against the triple store.

7.1 Experimental Setup

Our algorithms BGPtoSQL and SPARQLtoSQL were implemented in C++. The experiments were conducted on the PC with one 2.4 GHz Pentium IV CPU and 1024 MB of main memory operated by MS Windows XP Professional. Oracle9i Enterprise Edition Release 9.2.0.1.0 was used as a relational database back-end. The *SQL*Plus* utility was used to evaluate SQL queries over the database.

The timings reported below are the mean result from five or more trails with warm caches.

7.2 Experiment I: BGPtoSQL scalability on different graph patterns

We implemented a graph pattern generator which is capable to generate basic graph patterns encoded with the SPARQL syntax. For this experiment, we generated basic graph patterns that conform to the following three layouts (see Figure 7):

1. Chain – $|N| = |E| + 1$,
2. Complete binary tree – $|E| = 2^{h+1} - 2$, $|N| = |E| + 1$,
3. Dense graph – $|N| = \lceil\sqrt{|E|}\rceil$,

where h is the height of a tree (when applicable), $|E|$ and $|N|$ are the quantities of edges and nodes in a graph pattern, respectively.

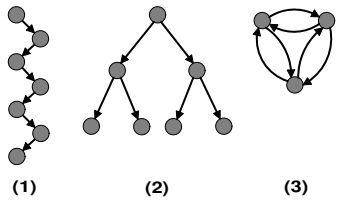

Figure 7: Graph pattern layouts: (1) chain, (2) complete binary tree, (3) dense graph

The BGPtoSQL performance for graph patterns of different sizes and layouts is reported in Table 1. The proposed algorithm showed to be very efficient – the translation of a graph pattern with less than 200 triple patterns took less than 0.001 sec. regardless of the graph pattern layout. The construction of a graph pattern model took considerably less time than the BGPtoSQL execution. Note that the graph pattern model construction showed to be the most efficient for dense graph patterns which can be explained by the relatively small cardinality of node set N. The BGPtoSQL performance was the best for chain graph patterns and the worst for dense graph patterns. However, the total time (sum of model construction time and SQL generation time) showed to be quite similar for all three layouts.

Table 1: BGPtoSQL scalability on different graph patterns (time shown in sec.)

Chain	$\|E\| = 254, \|N\| = 255$	$\|E\| = 2046, \|N\| = 2047$	$\|E\| = 4094, \|N\| = 4095$	$\|E\| = 8190, \|N\| = 8191$
Construct BGP	< 0.001	0.062	0.250	0.969
BGPtoSQL	< 0.001	0.238	0.972	5.672
Total	< 0.001	0.300	1.222	6.641
Binary tree	$\|E\| = 254, \|N\| = 255$	$\|E\| = 2046, \|N\| = 2047$	$\|E\| = 4094, \|N\| = 4095$	$\|E\| = 8190, \|N\| = 8191$
Construct BGP	< 0.001	0.047	0.219	0.813
BGPtoSQL	0.015	0.250	1.015	6.000
Total	0.015	0.297	1.234	6.813
Dense graph	$\|E\| = 254, \|N\| = 16$	$\|E\| = 2046, \|N\| = 46$	$\|E\| = 4094, \|N\| = 64$	$\|E\| = 8190, \|N\| = 91$
Construct BGP	< 0.001	0.016	0.078	0.344
BGPtoSQL	< 0.001	0.281	1.157	6.703
Total	< 0.001	0.297	1.235	7.047

7.3 Experiment II: SPARQLtoSQL scalability on different clause tree layouts

We generated SPARQL queries with optional graph patterns, such that their corresponding clause trees conform to the chain layout or the binary tree layout (see Figure 7). Each generated query contained one triple pattern in its `WHERE` clause and one triple pattern in its every `OPTIONAL` clause.

The SPARQLtoSQL performance for clause trees of different sizes and layouts is reported in Table 2, where $|E|$ and $|N|$ are the quantities of edges and nodes in a query clause tree. The proposed algorithm showed to be very efficient – the translation of queries with less than 50 `OPTIONAL` clauses (with one triple pattern in each) took less than 0.001 sec. regardless of the clause tree layout. The construction of a query model took less than 0.001 sec. in all our evaluations and, therefore, it did not influence the total execution time. The total time (sum of model construction time and SQL generation time) showed to be quite similar for the two tree layouts.

Table 2: SPARQLtoSQL scalability on different clause tree layouts (time shown in sec.)

Chain	$\|E\| = 62, \|N\| = 63$	$\|E\| = 126, \|N\| = 127$	$\|E\| = 254, \|N\| = 255$	$\|E\| = 510, \|N\| = 511$
Construct Q	< 0.001	< 0.001	< 0.001	< 0.001
SPARQLtoSQL	0.016	0.141	1.187	12.426
Total	0.016	0.141	1.187	12.426
Binary tree	$\|E\| = 62, \|N\| = 63$	$\|E\| = 126, \|N\| = 127$	$\|E\| = 254, \|N\| = 255$	$\|E\| = 510, \|N\| = 511$
Construct Q	< 0.001	< 0.001	< 0.001	< 0.001
SPARQLtoSQL	0.015	0.140	1.125	11.764
Total	0.015	0.140	1.125	11.764

7.4 Experiment III: Matching basic and optional graph patterns

To have a proof of concept and to get some insight on efficiency of SQL queries generated by our algorithms, we evaluated several such queries over the `Triples` table: `Triples(subject varchar2(110), predicate varchar2(50), object varchar2(520))`. The table was populated with 714,851 triples from the RDF representation of WordNet [13] (version: 1.2; author: Claudia Ciorascu). Note that the used RDF storage scheme is not very efficient [28] compared to some other storage schemes, and therefore we expect better performance for other storage schemes. Also, to avoid the overhead of returning multiple tuples to the client, we used aggregate functions in the outmost SQL `SELECT` clause list (e.g., each column name appears as `AVG(LENGTH(column))`).

The test queries, their patterns, variables, required SQL joins, result and execution time are shown in Table 3. The B^+-tree indexes were creates for each table column to evaluate queries Q3, Q4 and Q9. For other queries, the indexes were not used, since the overhead of accessing the indexes resulted in worse performance time. Queries Q1-Q7 matched basic graph patterns in the database. Queries Q1, Q2, Q5-Q7 whose final result contained considerable number of tuples showed to be quite expensive, especially Q2, Q6

[2]IJ stands for "inner join", and LOJ stands for "left outer join".

Table 3: SPARQL test queries

Query	**Patterns, variables**	**SQL joins**[2]	**Result** (tuples)	**Time** (sec.)
Q1: Find all resources of type *wn:Noun* `?a rdf:type wn:Noun .`	1 triple, 1 var	0	75,804	1.09
Q2: Find all triples whose subject is of type *wn:Noun* `?a rdf:type wn:Noun . ?a ?b ?c .`	2 triples, 4 vars, 3 distinct vars	1 IJ	385,544	40.04
Q3: Find all triples whose subject is a specific resource found by Q2	1 triple, 2 vars	0	4	<0.01
Q4: Find values of *rdf:type*, *wn:wordForm*, *wn:glossaryEntry* and *wn:hyponymOf* properties for a specific resource found by Q2 `wn:102040486 rdf:type ?a . ... wn:102040486 wn:hyponymOf ?d`	4 triples, 4 vars	3 IJs	1	<0.01
Q5: `?n1 wn:hyponymOf ?n2 .`	1 triple, 2 vars	0	90,267	1.09
Q6: `?n1 wn:hyponymOf ?n2 . ?n2 wn:hyponymOf ?n3 . ?n3 wn:hyponymOf ?n4 .`	3 triples, 6 vars, 4 disctinct vars	2 IJs	88,213	16.01
Q7: `?n1 wn:hyponymOf ?n2 . ?n2 wn:hyponymOf ?n3 . ... ?n6 wn:hyponymOf ?n7 .`	6 triples, 12 vars, 7 disctinct vars	5 IJs	67,788	42.03
Q8: Same triple patterns as in Q4, but each one is placed in a separate clause (`WHERE/OPTIONAL`), such that `W{O{O{O{}}}}`	3 `OPTIONAL`s	3 LOJs	1	8.06
Q9: Same triple patterns as in Q4, but each one is placed in a separate clause (`WHERE/OPTIONAL`), such that `W{O{}O{}O{}}`	3 `OPTIONAL`s	3 LOJs	1	<0.01
Q10: Same triple patterns as in Q6, but last two of them are placed in an `OPTIONAL` clause, such that `W{O{}}`	1 `OPTIONAL`	1 IJ 1 LOJ	96,765	16.00
Q11: Same triple patterns as in Q6, each one is placed in a separate clause (`WHERE/OPTIONAL`), such that `W{O{O{}}}`	2 `OPTIONAL`s	2 LOJs	96,804	>10 hours

and Q7 that required joins of large intermediate relations. On the other hand, low-selectivity queries (Q3 and Q4) were very fast even with the used storage scheme. Queries Q8-Q11 matched optional graph patterns in the database. The indexes benefited the performance of Q9 (without the indexes its performance was similar to Q8), but not Q8. Q10, derived from Q6, showed that placing some triple patterns of a basic graph pattern in an `OPTIONAL` clause, does not necessary result in worse performance time. Finally, Q11 showed to be extremely expensive (many hours of execution time). The divide-and-conquer strategy – breaking SQL query into several queries and storing their results in temporary tables – did not help. The Q11 evaluation in the MySQL 5.0.18 RDBMS showed even worse performance. The reason for such inefficient performance of Q11 is in the join condition which checks that certain columns must contain no `NULL` values. When the `NOT NULL` check is removed, the query finished in about 150 sec. producing the correct result. However, unlike Q11, the correctness of the result can not be always guaranteed without explicit `NULL` value checks. Note that Q8 also required `NOT NULL` checks, but Q8 is a low-selectivity query, which resulted in its satisfactory performance. The efficiency study of optional graph pattern queries that generate `NULL` value checks is a serious issue, which we leave as our future work.

8 Conclusions and Future Work

In this paper, we presented the SPARQL-to-SQL query translation under the optional graph pattern semantics. Specifically, we proposed: (i) a basic graph pattern translation algorithm, BGPtoSQL, that translates a basic graph pattern to its SQL equivalent; and based on BGPtoSQL, (ii) a semantics preserving SPARQL-to-SQL query translation algorithm, SPARQLtoSQL, for SPARQL queries that contain arbitrary complex optional graph patterns. Conclusions of our experimental study are:

- Our translation algorithms are efficient and scalable. For example, SPARQLtoSQL translated queries with less than 50 `OPTIONAL` clauses (with one triple pattern in each) in less than 0.001 sec. regardless of the clause tree layout.
- The preliminary study of generated SQL revealed that low-selectivity queries, whose intermediate and final results are small, have good performance. High-selectivity queries are more expensive and may become unsatisfactorily slow when contain many left outer joins and `NULL` value checks in their join conditions.

In the future, we will study the query optimization problem of both SPARQL and equivalent SQL queries. Also, the problem of other SPARQL constructs translation, such as `UNION` and `FILTER`, will be addressed.

Acknowledgements

We thank Shwetal Joshi, an Oracle DBA, for his assistance with Oracle tuning and useful comments on efficiency of SQL queries. We thank Richard Cyganiak for his constructive and encouraging comments.

References

[1] ARC SPARQL2SQL Rewriter. `http://www.appmosphere.com/pages/en-arc_sparql2sql_rewriter`.

[2] ARQ - A SPARQL processor for Jena. `http://jena.sourceforge.net/ARQ/`.

[3] KAON2. `http://kaon2.semanticweb.org/`.

[4] Rasqal RDF query library. `http://librdf.org/rasqal/`.

[5] RDF Primer. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/rdf-primer/`.

[6] RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/rdf-schema/`.

[7] RDFQL database command reference. `http://www.intellidimension.com/pages/rdfgateway/reference/db/default.rsp`.

[8] RDQL - A Query Language for RDF. W3C Member Submission, 9 January 2004. `http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/`.

[9] Sesame: RDF schema querying and storage. `http://www.openrdf.org/`.

[10] SPARQL Query Language for RDF. W3C Working Draft, 4 October 2006. `http://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004/`.

[11] sparql2sql a query engine for SPARQL over Jena triple stores. `http://jena.sourceforge.net/sparql2sql/`.

[12] User guide for Sesame. Updated for Sesame release 1.2.3. `http://www.openrdf.org/doc/sesame/users/index.html`.

[13] WordNet, a lexical database for the English language. `http://wordnet.princeton.edu/`.

[14] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[15] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *ISWC*, 2002.

[16] A. Chebotko, S. Lu, H. M. Jamil, and F. Fotouhi. Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical Report TR-DB-052006-CLJF. May 2006. `http://www.cs.wayne.edu/~artem/main/research/TR-DB-052006-CLJF.pdf`.

[17] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, 2005.

[18] R. Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170. 2005. `http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html`.

[19] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. In *SSWS*, 2005.

[20] E. Hung, Y. Deng, and V. S. Subrahmanian. RDF aggregate queries and views. In *ICDE*, 2005.

[21] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and Michel Scholl. RQL: A declarative query language for RDF. In *WWW*, 2002.

[22] Z. Pan and J. Heflin. DLDB: Extending relational databases to support Semantic Web queries. In *PSSS*, 2003.

[23] J. Perez, M. Arenas, and C. Gutierrez. *Semanticsof SPARQL.* `http://ing.utalca.cl/~jperez/papers/sparql_semantics.pdf`.

[24] J. Perez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *ISWC*, 2006.

[25] E. Prud'hommeaux. *Notes on Adding SPARQL to MySQL.* `http://www.w3.org/2005/05/22-SPARQL-MySQL/`.

[26] E. Prud'hommeaux. *Optimal RDF Access to Relational Databases.* `http://www.w3.org/2004/04/30-RDF-RDB-access/`.

[27] OpenLink Software. *Universal Server Platform for the Real-Time Enterprise.* `http://www.openlinksw.com/virtuoso/index.html`.

[28] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of RDF/S stores. In *ISWC*, 2005.

[29] R. Volz, D. Oberle, B. Motik, and S. Staab. KAON SERVER -A Semantic Web Management System. In *WWW, Alternate Tracks - Practice and Experience*, 2003.

[30] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *SWDB*, 2003.