Efficient Processing of RDF Queries with Nested Optional Graph Patterns in an RDBMS

Artem Chebotko, University of Texas - Pan American, USA
Shiyong Lu, Wayne State University, USA
Mustafa Atay, Winston-Salem State University, USA
Farshad Fotouhi, Wayne State University, USA

ABSTRACT

Relational technology has shown to be very useful for scalable Semantic Web data management. Numerous researchers have proposed to use RDBMSs to store and query voluminous RDF data using SQL and RDF query languages. In this article, we study how RDF queries with the so-called well-designed graph patterns and nested optional patterns can be efficiently evaluated in an RDBMS. We propose to extend relational databases with a novel relational operator, nested optional join (NOJ), that is more efficient than left outer join in processing nested optional patterns of well-designed graph patterns. We design three efficient algorithms to implement the new operator in relational databases: (1) nested-loops NOJ algorithm (NL-NOJ); (2) sort-merge NOJ algorithm (SM-NOJ); and (3) simple hash NOJ algorithm (SH-NOJ). Based on a real-life RDF dataset, we demonstrate the efficiency of our algorithms by comparing them with the corresponding left outer join implementations and explore the effect of join selectivity on the performance of our algorithms.

Keywords: Nested Optional Join; NOJ; Query Processing; RDBMS; RDF; Relational Join; Relational Operator; SPARQL; Semantic Web

INTRODUCTION

The Semantic Web (Berners-Lee, Hendler, & Lassila, 2001; Shadbolt, Berners-Lee, & Hall, 2006) 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. Semantic data are represented in resource description framework (RDF) (W3C, 2004a, 2004b), the standard language for annotating resources on the Web, and queried using the SPARQL (W3C, 2008) query language for RDF that has been recently proposed by the World Wide Web Consortium. RDF data are a collection of statements, called triples, of the form \( <s, p, o> \), where \( s \) is a subject, \( p \) is a predicate, and \( o \) is an object, and each triple states the relation between the subject and the object. Such collection 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 allows the specification of triple and graph patterns to be matched over RDF graphs.

Increasing amount of RDF data on the Web drives the need for its efficient and effective management. In this light, numerous researchers (Abadi, Marcus, Madden, & Hollenbach, 2007; Agrawal, Somani, & Xu, 2001; Alexaki, Christophides, Karvounarakis, & Plexousakis, 2001; Beckett & Grant, 2003; Broekstra, Kampman, & Harmelen, 2002; Erling, 2001; Harris & Gibbins, 2003; Ma, Su, Pan, Zhang, & Liu, 2004; Narayanan, Kurc, & Saltz, 2006; Pan & Heflin, 2003; Sintek & Kiesel, 2006; Stoffel, Taylor, & Hendler, 1997; Theoharis, Christophides, & Karvounarakis, 2005; Volz, Oberle, Motik, & Staab, 2003; Wilkinson, 2006; Wilkinson, Sayers, Kuno, & Reynolds, 2003) 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.

An important class of graph patterns that are mostly common in RDF queries in practice is the so-called well-designed graph patterns (Perez, Arenas, & Gutierrez, 2006a). A well-designed graph pattern (\( gp \)) can contain many arbitrary optional graph patterns that can be nested in each other as in the following equation:

\[
gp_1\OPT(gp_2\OPT(gp_3\OPT(... \OPT(gp_n))))
\]

(1)

where each \( gp_1, gp_2, gp_3, ..., gp_{n-1}, gp_n \) can be a basic graph pattern (set of triple patterns) or another graph pattern with optional subpatterns such as (1), \( \OPT \) indicates an optional graph pattern that follows it, and parentheses define the order of evaluation. In (1), \( gp_2, gp_3, ..., gp_{n-1}, gp_n \) are optional graph patterns, and each \( gp_i, i \geq 3 \), is a nested optional graph pattern with respect to \( gp_{i-1} \). By the definition of a well-designed graph pattern, the following property for \( gp \) holds: for any subpattern \( (gp_x\OPT(gp_y)) \) in \( gp \), if a variable \(?v\) occurs both outside this subpattern and inside \( gp_y \), then \(?v\) also occurs in \( gp_x \). The formal semantics of the well-designed graph patterns with nested optional patterns is defined by Perez et al. (2006a) and W3C (2008). Informally, it can be summarized as follows:

- **Basic semantics of optional graph patterns.** The evaluation of an optional graph pattern is not obligated to succeed, and in the case of failure, its variables are unbound. For example, in (1), \( gp_n \) does not have to succeed for \( gp_{n-1} \) to succeed.
- **Semantics of shared variables in optional graph patterns.** In general, shared variables must be bound to the same values. Variables can be shared
among subjects, predicates, objects, and across each other. For example, in (1), if a variable \( ?v \) occurs both inside \( gp_1 \) and \( gp_2 \), it must be bound to the same value in both graph patterns.

- **Semantics of nested optional graph patterns.** Before a nested optional graph pattern can succeed, all containing optional graph patterns must have succeeded. For example, in (1), \( gp_3 \) corresponds to an optional graph pattern nested inside another optional graph pattern \( gp_2 \), and \( gp_3 \) can only succeed if \( gp_2 \) succeeds.

Therefore, a well-designed graph pattern \( (gp) \) as in (1) can have \( n - 2 \) nested optional graph subpatterns \( gp_3, \ldots, gp_{n-1}, gp_n \), and an efficient evaluation of these nested patterns is very important. While the literature on the SPARQL-to-SQL translation is abundant (see the related work section), a few researches (Chebotko, Lu, Jamil, & Fotouhi, 2006; Cyganiak, 2005) study the translation of RDF queries with nested optional patterns. In their work, the handling of these three semantics in a relational database relies on the use of the left outer join (LOJ) defined in the relational algebra and SQL: (1) basic semantics of optional graph patterns is captured by an LOJ; (2) semantics of shared variables is treated with the conjunction of equalities of corresponding relational attributes in the LOJ condition; (3) semantics of nested optional graph patterns is preserved by the \textit{NOT NULL} check in the LOJ condition for one of the attributes/variables that correspond to the parent (containing pattern) of a nested pattern. In the following, we present our running example to illustrate the translation of a SPARQL query with a nested optional graph pattern into a relational algebra expression, in which an LOJ is used for implementing nested optional graph patterns; the example motivates the introduction of a new relational operator for a more efficient implementation.

**Example 1 (Sample SPARQL query and its relational equivalent).** Consider the RDF graph presented in Figure 1(a). The graph describes academic relations among professors and graduate students in a university. The RDF schema defines two concepts/classes (Professor and GradStudent) and two relations/properties (hasAdvisor and hasCoadvisor). Each relation has the GradStudent class as a domain and the Professor class as a range. Additionally, two instances of Professor, two instances of GradStudent, and relations among these instances are defined as shown in the figure.

We design an RDF query that returns (1) every graduate student in the RDF graph; (2) the student’s advisor if this information is available; and (3) the student’s coadvisor if this information is available and if the student’s advisor has been successfully retrieved in the previous step. In other words, the query returns students and as many advisors as possible; there is no point to return a coadvisor if there is even no advisor for a student. The SPARQL representation of the query is as follows:

```
01 SELECT ?stu ?adv ?coadv
02 WHERE {
03  ?stu rdf:type :GradStudent . /* R1(stu) */
04  OPTIONAL {
05   ?stu :hasAdvisor ?adv . /* R2(stu,adv) */
06  OPTIONAL {
07    ?stu :hasCoadvisor ?coadv ./* R3(stu,coadv) */
08  } } }
```
The query has three variables: ?stu for the student, ?adv for the advisor, and ?coadv for the coadvisor. There are two OPTIONAL clauses, where the innermost one is the nested OPTIONAL clause. The graph pattern in the WHERE clause is well-designed and corresponds to the pattern (gp_1 OPT (gp_2 OPT gp_3)), where \( gp_1 = \{ ?stu \text{ rdf:type :GradStudent} \} \), \( gp_2 = \{ ?stu \text{ hasAdvisor ?adv} \} \), and \( gp_3 = \{ ?stu \text{ hasCoadvisor ?coadv} \} \). Variable \( ?stu \) is a shared variable that occurs in \( gp_1 \), \( gp_2 \), and \( gp_3 \).

To translate this SPARQL query into an equivalent relational query, we use our translation strategy (Chebotko et al., 2006) as follows. Matching triples for the triple patterns \( gp_1 \), \( gp_2 \), and \( gp_3 \) are retrieved into relations \( R_1 \), \( R_2 \), and \( R_3 \) respectively. Note that the triple patterns are annotated with the corresponding relations and relational schemas in the previous SPARQL query. Then the equivalent relational algebra representation is

\[
R_4 = \pi_{R_4.stu,R_2.adv} (R_1 \bowtie_{R_1.stu=R_2.stu} R_2),
\]

\[
R_{res} = \pi_{R_4.stu,R_4.adv,R_3.coadv} (R_4 \bowtie_{R_4.stu=R_3.stu} R_3 \text{ adv IS NOT NULL } R_3).
\]
Each \textit{OPTIONAL} clause corresponds to the left outer join (\(=\bowtie\)), shared variable \(?stu\) participates in the join conditions, and the nested \textit{OPTIONAL} implements the \textit{NOT NULL} check on the \textit{adv} attribute to ensure that its parent clause has indeed succeeded. The graphical representation of the relational query is shown in Figure 1(b); the projection operators are not shown for ease of presentation.

The running example motivates our research. The following is our insight to how the LOJ-based query in Figure 1(b) wastes some computations: (1) Based on the result of the first LOJ and the semantics of the nested optional graph pattern, we know that the \textit{NULL} padded tuple (\textit{Natalia}, \textit{NULL}) will also be \textit{NULL} padded in the second LOJ. After all, there is no need for this tuple to participate in the second LOJ condition. (2) On the other hand, we know that the successful match in the tuple (\textit{Artem}, \textit{Shiyong}) contains no \textit{NULL}s. There is no need to apply the \textit{NOT NULL} check to this tuple.

In this article, we propose to extend relational technology with an innovative relational operator that naturally supports the nested optional pattern semantics of well-designed graph patterns to enable their efficient processing in relational databases. The main contributions of our work include the following:

- We propose to extend relational databases with a novel relational operator, \textit{nested optional join (NOJ)}, that is more efficient than left outer join in processing nested optional graph patterns of well-designed graph patterns. The computational advantage of NOJ over the currently used LOJ-based implementations comes from the two superior characteristics of NOJ: (1) NOJ allows the processing of the tuples that are guaranteed to be \textit{NULL} padded very efficiently (in linear time) and (2) NOJ does not require the \textit{NOT NULL} check to return correct results.

- Based on a real-life RDF dataset, we demonstrate the efficiency of our algorithms by comparing them with the corresponding left outer join implementations. The experimental results are very promising; for RDF query processing with RDBMSs, NOJ is a favorable alternative to the LOJ-based evaluation of nested optional patterns in well-designed graph patterns.

- Based on both our theoretical analysis and empirical study, we give our recommendations for the use of the presented NOJ algorithms depending on the selectivity factor of the join.

This work extends our conference paper (Chebotko, Atay, Lu, & Fotouhi, 2007) in several directions. First, we address the changes in the SPARQL semantics, positioning our work for the evaluation of well-designed graph patterns that are most commonly used in real-life RDF queries. Second, we present two new algorithms for nested optional join implementation; namely, a sort-merge NOJ algorithm, SM-NOJ, and a simple hash NOJ algorithm,
SH-NOJ. These algorithms have better time complexity and show better performance on our test queries than our first proposed algorithm NL-NOJ (nested-loops NOJ algorithm). Third, we provide extensive details for our conducted experiments, including explanation of datasets, test queries, implementations, and observed results. Fourth, we conduct an additional performance study of the NOJ algorithm behavior with respect to varying join selectivity factor, giving our recommendations on the applicability of the three NOJ algorithms. Fifth, we implement our algorithm NL-NOJ in an existing RDBMS, MySQL, and compare the performance of our test queries evaluated with NL-NOJ and with MySQL’s nested-loops join algorithm. Finally, we compare the performance of our RDBMS NOJ implementation with RDF stores Sesame and Jena.

NESTED OPTIONAL JOIN OPERATOR

In this section, we present our nested optional join operator that is used to evaluate nested optional patterns of well-designed graph patterns in relational databases and highlight its advantages over the left outer join.

The operands of NOJ are twin relations instead of conventional relations. The notion of twin relation is introduced as follows.

**Definition 2 (Twin Relation).** A twin relation, denoted as \((R_b, R_o)\), is a pair of conventional relations with identical relational schemas and disjoint sets of tuples. The schema of a twin relation is denoted as \(\xi(R_b, R_o)\). \(R_b\) with the schema \(\xi(R_b, R_o)\) and called a base relation, and \(R_o\) with the schema \(\xi(R_b, R_o)\) is called an optional relation. A distinguished tuple \(r_\xi(R_b, R_o)\) is defined as a tuple of \(\xi(R_b, R_o)\) in which each attribute takes a NULL value. ◊

Intuitively, a base relation is used to store tuples that have a potential to satisfy a join condition of a nested optional join. An optional relation is used to store tuples that are guaranteed to fail a join condition of a nested optional join. We incorporate the twin relation into the relational algebra by introducing the following additional operators, \(\cup\) and \(\hat{\cup}\), such that:

- \(\cup(R_b, R_o) = R_b \cup R_o\) and
- \(\hat{\cup}(R) = (R, \emptyset)\), where \(\emptyset\) is an instance of empty relation with the same relational schema of \(R\).

Note that \(\hat{\cup}\) is not a reversed operator of \(\cup\), because \(\hat{\cup}(R_b, R_o) \neq (R_b, R_o)\) in general.

We also extend the projection and selection operators to a twin relation as \(\pi[(R_b, R_o)] = (\pi[R_b], \pi[R_o])\) and \(\sigma[(R_b, R_o)] = (\sigma[R_b], \sigma[R_o])\), respectively. The definition of a complete algebra for a twin relation is not our focus in this article; \(\pi\) and \(\sigma\) are sufficient for our running example and experimental study and, as we believe, for most SPARQL-to-SQL translations.

In the following, we define a novel relational operator, nested optional join, using the tuple calculus.

**Definition 3 (Nested Optional Join).** A nested optional join of two twin relations, denoted as \(\equiv^{\exists=\xi}\), yields a twin relation, such that \((R_b, R_o) \equiv^{\exists=\xi} \pi_{r(a) \leftarrow (b)}(S_b, S_o) = (Q_b, Q_o)\), where \(Q_b = \{t \mid t = rs \land r \in R_b \land (s \in S_b \lor s \in S_o)\}\).
∈ S_o) ∧ r(a) = s(b)} and \( Q_o = \{ t \mid t = rn ∧ (r ∈ R_o ∨ (r ∈ R_b ∧ °andr(a) = s(b)))\} \), where \( r(a) = s(b) \) is a join predicate, \( r(a) ⊆ ξ(R_b, R_o) \) and \( s(b) ⊆ ξ(S_b, S_o) \) are join attributes, \( n = n(\xi(S_b, S_o) − \diamond) \).

In other words, the result base relation \( Q_b \) contains tuples \( t \) made up of two parts, \( r \) and \( s \), where \( r \) must be a tuple in relation \( R_b \) and \( s \) must be a tuple in \( S_b \) or \( S_o \). In each tuple \( t \), the values of the join attributes \( t(a) \), belonging to \( r \), are identical in all respects to the values of join attributes \( t(b) \), belonging to \( s \). The result optional relation \( Q_o \) contains tuples \( t \) made up of two parts, \( r \) and \( n \), where \( r \) must be a tuple in \( R_o \) with no other conditions enforced, or \( r \) must be a tuple in \( R_b \) and there must not exist a tuple \( s \) in \( S_b \) or \( S_o \) that can be combined with \( r \) based on the predicate \( r(a) = s(b) \).

The graphical illustration of the NOJ operator is shown in Figure 2. Note how well it emphasizes one of the advantages of NOJ: the flow of tuples from \( R_o \) to \( Q_o \) bypasses the join condition and does not interact with tuples from any other relation. Obviously, the behavior of this flow can be implemented to have linear time performance in the worst case—the property that is, in general, not available in the LOJ implementations. The second important advantage of NOJ—no need for the NOT NULL check—is discussed in the following example that describes the translation of our sample SPARQL query into a relational algebra expression with our extensions.

**Example 4 (Evaluation of the sample SPARQL query using NOJs).** We use the same RDF graph presented in Figure 1(a) and the SPARQL query described in Example 1. The translation strategy is similar to the one illustrated in Example 1, except that we use NOJ instead of LOJ. Matching triples for the triple patterns \(?stu rdf:type :GradStudent, \ ?stu :hasAdvisor ?adv, \ and \ ?stu :hasCoadvisor ?coadv\) are retrieved into relations \( R_1, R_2, \) and \( R_3 \), respectively. Then the equivalent relational algebra representation using NOJ is:

\[
(R_b, R_o^1) = ζ(R_1), (R_b^2, R_o^2) = ζ(R_2),
(R_b^3, R_o^3) = ζ(R_3)
\]

\[
(R_b^4, R_o^4) = π_{(R_b^1, R_o^1),stu=(R_b^2, R_o^2)}((R_b^1, R_o^1)),
\]

\[
\equiv ξ_{(R_b^1, R_o^4),stu=(R_b^2, R_o^2)}((R_b^2, R_o^3)),
\]

\[
(R_b^{res}, R_o^{res}) = π_{(R_b^4, R_o^4),stu=(R_b^4, R_o^4),adv,(R_b^3, R_o^3),coadv}((R_b^4, R_o^4))
\]
\[ \exists \bowtie (R^1_{b}, R^3_{o}).stu = (R^1_{b}, R^3_{o}).stu (R^2_{b}, R^3_{o}), \]

\[ R_{res} = \bowtie (R^res_{b}, R^res_{o}). \]

The graphical representation of the relational query is shown in Figure 3; the conversion and projection operators are not shown for ease of presentation. Note that this query does not contain the **NOT NULL** check, because all the tuples that have not succeeded in the first join are padded with **NULL** values and stored into the optional relation \( R^3_{o} \); the tuples of \( R^4_{o} \) bypass the second join condition and are copied directly to \( R^res_{o} \) with additional **NULL**-padding. ◊

Therefore, NOJ is superior to LOJ when we apply them to translate SPARQL-nested optional patterns of well-designed graph patterns to relational queries. The main advantages of NOJ are (1) NOJ allows the processing of the tuples that are guaranteed to be **NULL** padded very efficiently (in linear time); (2) NOJ does not require the **NOT NULL** check to return correct results; and (3) NOJ can significantly simplify the translation of RDF queries with well-designed graph patterns into relational algebra, eliminating the need to choose a relational attribute for the **NOT NULL** check\(^1\) and, in some cases (Chebotko et al., 2006) when such an attribute cannot be chosen from available ones, the need to introduce a new variable or even a new triple pattern into a SPARQL query. Our performance study showed that these advantages bring substantial speedup to the query evaluation.

**NESTED OPTIONAL JOIN ALGORITHMS**

Previously, we defined NOJ through the tuple calculus, but it is also possible to express the NOJ result relations using standard operators of the relational algebra:

\[ Q_{b} = R_{b} \bowtie_{(a)\neq(stu)} (S_{b} \cup S_{o}) \]  
\[ Q_{o} = R'_{o} \cup \left( (R_{b} \bowtie_{(a)\neq(stu)} (S_{b} \cup S_{o})) - (R_{b} \bowtie_{(a)\neq(stu)} (S_{b} \cup S_{o}) \right), \]  
where \( R'_{o} \) is **NULL**-padded to schema \( \xi_{o}(R_{b}, R_{o}) \cup \xi_{o}(S_{b}, S_{o}) \). However, it should be evident that this direct translation will be inefficient if implemented. Therefore, in this section, we design our own algorithms to implement NOJ in a relational database. Our algorithms, **NL-NOJ**, **SM-NOJ**, and **SH-NOJ**, employ the classic methods used to implement relational joins: nested-loops,
sort-merge, and hash-based join methods, respectively.

**Nested-Loops Nested Optional Join Algorithm**

The simplest algorithm to perform the NOJ operation is the nested-loops NOJ algorithm, denoted as **NL-NOJ** (see Figure 4). The algorithm is self-descriptive. For each tuple \( r \) in relation \( R_b \), twin relation \((S_b, S_o)\) is scanned (lines 05-07), and tuples satisfying the join condition are merged into \( Q_b \) (lines 08-11). Tuples in \( R_b \) that have no matching tuples in \((S_b, S_o)\) are NULL padded and placed in \( Q_o \) (lines 13-15). Finally (lines 17-19), tuples in \( R_o \) are NULL padded and placed in \( Q_o \).

The results of our complexity and applicability analysis are

- **NL-NOJ complexity**: \( \Theta(|R_b| \times (|S_b| + |S_o|) + |R_o|) \).
- **NL-NOJ applicability**: NOJs with high selectivity factors (see our performance study for more details).

The comprehensive analysis of the performance and applicability of the nested-loops join method is presented by Mishra and Eich (1992). In the join processing literature, there is a number of optimizations on the nested-loops join method that are also applicable to **NL-NOJ** (e.g., the block nested-loops join method (Elmasri & Navathe, 2004; Kifer, Bernstein, & Lewis, 2006) and “rocking” the inner relation optimization (Kim, 1980) that reduce the number of I/O operations).

**Sort-merge Nested Optional Join Algorithm**

The sort-merge NOJ algorithm, **SM-NOJ**, is shown in Figure 5. **SM-NOJ** is executed in three stages. First (lines 05-07), relations \( R_b \) and \( S \) are sorted on the join attributes, where

---

**Figure 4. Algorithm NL-NOJ**

```plaintext
01 Algorithm: NL-NOJ
02 Input: twin relations \((R_b, R_o)\) and \((S_b, S_o)\)
03 Output: twin relation \((Q_b, Q_o) = (R_b, R_o) \bowtie_{(a,b)} (S_b, S_o)\)
04 Begin
05 For each tuple \( r \in R_b \) do
06    pad = true
07    For each tuple \( s \in (S_b, S_o) \) do
08        If \( r(a) = s(b) \) then
09            place tuple \( rs \) in relation \( Q_s \)
10            pad = false
11        End If
12    End For
13    If pad then
14        place tuple \( n_{\text{null}} \) in relation \( Q_o \)
15    End If
16    For each tuple \( r \in R_o \) do
17        place tuple \( m_{\text{null}} \) in relation \( Q_o \)
18    End For
19 Return \((Q_b, Q_o)\)
20 End Algorithm
```
$S$ contains all tuples of $(S_b, S_o)$. Second (lines 08-36), $R_b$ and $S$ are scanned in the order of the join attributes, and tuples satisfying the join condition are merged into $Q_b$; tuples in $R_b$ that have no matching tuples in $S$ are $NULL$ padded and placed in $Q_o$. The scanning employs backtracking (lines 22-34), such that if $r$ in $R_b$ matches consecutive tuples in $S$, then $backtrack$ remembers the first matching tuple (line 24), and if $r$ and the next tuple after $r$ have the same values of the join attributes (lines 29-31), then the scanning of $S$ resumes from the tuple restored from $backtrack$ (line 34). At the final third stage (lines 37-39), the tuples in $R_o$ are $NULL$ padded and placed in $Q_o$.

The results of our complexity and applicability analysis are

**Figure 5. Algorithm SM-NOJ**

```
01 Algorithm: SM-NOJ
02 Input: twin relations ($R_o$, $R$) and ($S_b$, $S$)
03 Output: twin relation ($Q_o$, $Q$) = ($R_o$, $R$) ≡ $\bowtie_{r(a)=s(b)} (S_b, S_o)$
04 Begin
05 Sort $R$ on $r(a)$
06 Let $S = S_b \cup S_o$
07 Sort $S$ on $s(b)
08 r = first$ tuple of $R$
09 s = first$ tuple of $S$
10 While $r \neq EOF$ do
11     If $s = EOF$ then
12         place tuple $(r, s)$ in relation $Q_o$
13         $r = next$ tuple after $r$ of $R$
14     Else
15         While $r \neq EOF$ and $r(a) < s(b)$ do
16             place tuple $(r, s)$ in relation $Q_o$
17             $r = next$ tuple after $r$ of $R$
18         End While
19         While $s \neq EOF$ and $s(b) < r(a)$ do
20             $s = next$ tuple after $s$ of $S$
21         End While
22     back = false
23     If $r \neq EOF$ and $r(a) = s(b)$ then
24         backtrack = $s$
25         While $s \neq EOF$ and $s(b) = r(a)$ do
26             place tuple $(r, s)$ in relation $Q_o$
27             $s = next$ tuple after $s$ of $S$
28         End While
29     If $r$ and next tuple after $r$ of $R$ have the same values of $r(a)$ then
30         back = true
31     End If
32     $r = next$ tuple after $r$ of $R$
33     End If
34     If back then $s = backtrack$ End If
35     End If
36 End While
37 For each tuple $r \in R$ do
38     place tuple $(r, s)$ in relation $Q_o$
39 End For
40 Return ($Q_o$, $Q$)
41 End Algorithm
```
• **SM-NOJ complexity**: $\Omega(|R_b| \times \log|R_b| + (|S_b| + |S_o|) \times \log(|S_b| + |S_o|) + |R_o|), O(|R_b| \times (|S_b| + |S_o|) + |R_o|)$. The best case performance is achieved when there is no backtracking or, in other words, either $R_b$ or $S$ has no multiple tuples, such that these tuples have the same values of the join attributes and have at least one matching tuple in $S$ or $R_o$, respectively. The more backtracking is involved, the worse the performance of SM-NOJ. The worst case performance occurs when all tuples in both $R_b$ and $S$ have the same values of the join attributes, such that for any $r \in R_b$ and $s \in S$, $r(a) = s(b)$ is always satisfied. In the cases close to the worst case, SM-NOJ does a comparable number of computations to those of NL-NOJ and additionally sorts the relations and requires some extra operations that implement the backtracking mechanism; in the worst case, the relations are already sorted.

• **SM-NOJ applicability**: SM-NOJ is the best choice when NL-NOJ or SH-NOJ is not selected as the best performer; NOJs with median selectivity factors (see our performance study for more details).


**Simple Hash-Nested Optional Join Algorithm**

The simple hash NOJ algorithm, SH-NOJ, is presented in Figure 6. The algorithm uses a hash function $h$ to hash the tuples of the twin relation $(S_b, S_o)$ to a hash table $H$ based on the values of the join attributes (lines 05-09). A perfect hash function hashes tuples with different values of the join attributes to different buckets; however, in practice, such tuples may end up in the same bucket. Then for each tuple $r$ in $R_b$, the hash value of the join attributes is computed using the same hash function $h$ (line 12). If $r$ hashes to a nonempty bucket of $H$, then all the tuples in the bucket are compared with $r$ and merged to $Q_b$ if the join condition is satisfied (lines 12-15) or NULL padded and copied to $Q_o$ (lines 16-18) otherwise. Finally, the tuples in $R_o$ are NULL padded and placed in $Q_o$ (lines 20-22).

Note that the hash table should ideally be created for the (twin) relation with the fewest distinct values of the join attributes. When this information is not available, the hash table is usually created for the smallest of the (twin) relations $R_b$ and $(S_b, S_o)$ (Mishra & Eich, 1992).

The results of our complexity and applicability analysis are:

• **SH-NOJ complexity**: $\Omega(|R_b| + |R_o| + |S_b| + |S_o|), O(|S_b| \times (|S_b| + |S_o|) + |R_o|)$, depends on the efficiency of a hash function $h$. The linear performance is achieved for joins with empty results or joins with low selectivity factors. The higher the selectivity is, the slower SH-NOJ performs. In the worst case, when all tuples of $R_b$ and $(S_b, S_o)$ hash to the same bucket of the hash table, the algorithm has the quadratic performance.

• **SH-NOJ applicability**: NOJs with low selectivity factors (see our performance study for more details).

The comprehensive analysis of the performance and applicability of the simple hash join method is presented by Mishra and Eich (1992). The simple hash join method (DeWitt, Katz, Olken, Shapiro, Stonebraker, & Wood, 1984) that is used
Figure 6. Algorithm SH-NOJ

01 Algorithm: SH-NOJ
02 Input: twin relations \((R_b, R_o)\) and \((S_b, S_o)\)
03 Output: twin relation \((Q_b, Q_o) = (R_b, R_o) \bowtie (S_b, S_o)\)
04 Begin
05 Let \(h\) be a hash function and \(H\) be an empty hash table
06 For each tuple \(s \in (S_b, S_o)\) do
07 hash on join attributes \(s(b)\): \(h(s(b))\)
08 place \(s\) in the corresponding bucket of hash table \(H\): \(H\{h(s(b))\} += s\)
09 End For
10 For each tuple \(r \in R_s\) do
11 pad = true
12 For each tuple \(s \in H\{h(r(a))\}\) and \(r(a) = s(b)\) do
13 place tuple \(rs\) in relation \(Q_s\)
14 pad = false
15 End For
16 If \(pad\) then
17 place tuple \(m\) with \(s\) in relation \(Q_o\)
18 End If
19 End For
20 For each tuple \(r \in R_o\) do
21 place tuple \(m\) with \(r\) in relation \(Q_o\)
22 End For
23 Return \((Q_b, Q_o)\)
24 End Algorithm

Experimental Setup

For the first five experiments, we used our in-memory relational database. We implemented in-memory representations of a relation and a twin relation, such that each relation was represented by a doubly-linked list of tuples, where each tuple corresponded to an array of pointers to attribute data values, and each twin relation was represented by pointers to two conventional relations. The memory to store relations and their tuples was allocated dynamically in the heap.

Our algorithms NL-NOJ, SM-NOJ, and SH-NOJ were implemented in C++. To compare the performance of queries evaluated with our algorithms and corresponding left outer join algorithms, we implemented nested-loops LOJ (NL-LOJ), sort-merge LOJ (SM-LOJ), and simple hash LOJ (SH-LOJ) algorithms (see Mishra & Eich, 1992).
For the last two experiments, which required implementing NOJ in an RD-BMS, we used MySQL 5.0. MySQL was selected because it is currently the most popular open source database system. For the evaluation of left outer joins, MySQL employed an optimized variation of a nested-loops join algorithm, which we denoted as NL-LOJ-MySQL. In particular, nested joins of multiple relations were conducted concurrently, such that if \( k \) relations were joined, the join algorithm used \( k \) nested loops to compute one resulting tuple at a time (MySQL, 2008). Given a query with nested LOJs, MySQL did not perform any join reordering, but used B+ tree indexes whenever applicable.

Our algorithm NL-NOJ was implemented within the MySQL relational engine. We denoted this implementation as NL-NOJ-MySQL and compared its performance to NL-LOJ-MySQL using our test queries. To keep with the spirit of conventional relational join algorithms, NL-NOJ-MySQL was only passed two names of relations, \( R \) and \( S \), which served as a prefix to the names of the to-be-joined twin relations \((R_b, R_o)\) and \((S_b, S_o)\). For the very first (not nested) join in a query, when only conventional relations were available, \( R_2 = R \) and \( S_2 = S \), while \( R_o \) and \( S_o \) were considered to be empty. Even though each join result was a twin relation, this relation was represented by a single name prefix. Finally, the result of the last join in a query was combined into a single output relation. This way, we were able to completely hide twin relations outside of a query.

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.

The timings reported in our experiments are the mean result from five or more trials with warm caches.

**Dataset**

We conducted the experiments using the OWL representation of WordNet (Ciorascu, 2003), a lexical database for the English language, which organizes English words into synonym sets according to part of speech (e.g., noun, verb, etc.) and enumerates linguistic relations between these sets. In the WordNet.OWL, each part of speech is modeled as an \( \textit{owl:Class} \), and each linguistic relation is modeled as an \( \textit{owl:ObjectProperty} \), \( \textit{owl:DatatypeProperty} \), \( \textit{owl:TransitiveProperty} \), or \( \textit{owl:SymmetricProperty} \). The simplified WordNet ontology is illustrated in Figure 7. The figure does not include some classes (e.g., \( \textit{wn:Nouns_and_Verbs} \)) and properties (e.g., \( \textit{wn:mMeronym} \)) that are not essential for the understanding of the dataset and the experiments.

The relevant statistics for the WordNet dataset is shown in Table 1. For example, WordNet.OWL contains 251,726 triples involving \( \textit{rdf:type} \) as the predicate, and 140,470 of them have \( \textit{wn:WordObject} \) as the object.

**Relational Storage of RDF Data**

We stored the WordNet dataset into our in-memory relational database using \textit{property relations}, where a property relation (e.g., \( p(s,o) \)) was created for each property \( p \) in the ontology and stored subjects \( s \) and objects \( o \) related by this property in the RDF dataset.

To store the WordNet dataset into MySQL, we employed our relational RDF store \textit{RDFProv} (Chebotko, Fei, Lin, Lu, & Fotouhi, 2007; Chebotko, Fei, Lu, & Fotouhi, 2007). Based on the WordNet
ontology, *RDFProv* generated a database schema, shredded RDF triples into tuples, and populated the relations. While *RDFProv* generated several kinds of relations, all the joins in our experiments were performed on property relations. In addition, *RDFProv* created B+ tree indexes on every relational attribute of each property relation.

**Test Queries**

We chose 14 SPARQL queries to evaluate in our experiments based on the following criteria: (1) queries should contain well-designed graph patterns; (2) most queries should have nested optional graph patterns; (3) the input, intermediate, and output (twin) relations involved in the query evaluation should fit into the main memory; and (4) some queries should have common patterns to reveal performance changes with increasing complexity of the queries.

The test queries are listed in Table 2, where *W* stands for *WHERE* and *O* stands for *OPTIONAL*. The SPARQL *SELECT* clause

---

### Table 1. Property and resource statistics of WordNet

<table>
<thead>
<tr>
<th>Property</th>
<th>Count</th>
<th>Resource</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>251,726</td>
<td>WordObject</td>
<td>140,470</td>
</tr>
<tr>
<td>wordForm</td>
<td>195,802</td>
<td>Noun</td>
<td>75,804</td>
</tr>
<tr>
<td>glossaryEntry</td>
<td>111,223</td>
<td>Verb</td>
<td>13,214</td>
</tr>
<tr>
<td>hyponymOf</td>
<td>90,267</td>
<td>AdjectiveSatellite</td>
<td>11,231</td>
</tr>
<tr>
<td>similarTo</td>
<td>22,494</td>
<td>Adjective</td>
<td>7,345</td>
</tr>
<tr>
<td>antonymOf</td>
<td>7,115</td>
<td>Adverb</td>
<td>3,629</td>
</tr>
<tr>
<td>Others</td>
<td>36,225</td>
<td>Others</td>
<td>33</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>714,852</td>
<td><strong>Total</strong></td>
<td>251,726</td>
</tr>
</tbody>
</table>
is omitted for brevity, and the projection includes all distinct variables of the query. The relevant characteristics of the test queries, such as the number of required joins (either nested optional or left outer joins) and the cardinality of participating twin relations (input, intermediate, output), are shown in Table 3. The cardinalities are presented for twin relations as \((\text{Cardinality}_b, \text{Cardinality}_o)\); the corresponding cardinalities of conventional relations that participate in the left outer joins can be easily derived as \(\text{Cardinality}_b + \text{Cardinality}_o\).

Q1 is interesting because the cardinalities of participating (twin) relations are relatively small. Both Q1 and Q2 have only one optional graph pattern and therefore require single join. This implies that the nested optional join has no advantages over the left outer join in Q1 and Q2, since there are no nested optional patterns in the queries. Queries Q2-Q4 and similarly Q5-Q7 are related and can be derived from each other. Q7 stands out from Q6 and Q5 in considerably larger values of \(\text{Cardinality}_o\) in the intermediate twin relations. Queries Q8-Q14 resemble corresponding queries Q1-Q7 but substitute all the occurrences of one of the variables with a Uniform Resource Locator (URL). While Q1-Q7 involve quite expensive joins, Q8-Q14 are more specific and, in our setting, can take an advantage of database indexes. In addition, Q8-Q14 achieve the effect that all intermediate and output twin relations contain only a few tuples in their base relations, while optional relations are large.

An important characteristic of the test queries is that they only involve joins, whose selectivity factors are less than

<table>
<thead>
<tr>
<th>#</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>(\text{W} { ?a \text{ rdf:} \text{type} :\text{Adjective} \ \text{O}{ ?a \text{ :} \text{similarTo} \ ?b }} )</td>
</tr>
<tr>
<td>Q2</td>
<td>(\text{W} { ?a \text{ rdf:} \text{type} \ ?b \ \text{O}{ ?c \text{ :} \text{wordForm} \ ?a }} )</td>
</tr>
<tr>
<td>Q3</td>
<td>(\text{W} { ?a \text{ rdf:} \text{type} \ ?b \ \text{O}{ ?c \text{ :} \text{wordForm} \ ?a \ \text{O}{ ?c \text{ :} \text{glossaryEntry} \ ?d }}} )</td>
</tr>
<tr>
<td>Q4</td>
<td>(\text{W} { ?a \text{ rdf:} \text{type} \ ?b \ \text{O}{ ?c \text{ :} \text{wordForm} \ ?a \ \text{O}{ ?c \text{ :} \text{glossaryEntry} \ ?d \ \text{O}{ ?c \text{ :} \text{hyponymOf} \ ?e }}} )</td>
</tr>
<tr>
<td>Q5</td>
<td>(\text{W} { ?n1 \ \text{hyponymOf} \ ?n2 \ \text{O}{ ?n2 \ \text{hyponymOf} \ ?n3 \ \text{O}{ ?n3 \ \text{hyponymOf} \ ?n4 }}} )</td>
</tr>
<tr>
<td>Q6</td>
<td>(\text{W} { ?n1 \ \text{hyponymOf} \ ?n2 \ \text{O}{ ?n2 \ \text{hyponymOf} \ ?n3 \ \text{O}{ ?n3 \ \text{hyponymOf} \ ?n4 \ \text{O}{ ?n4 \ \text{hyponymOf} \ ?n5 \ \text{O}{ ?n5 \ \text{hyponymOf} \ ?n6 \ \text{O}{ ?n6 \ \text{hyponymOf} \ ?n7 }}}}} )</td>
</tr>
<tr>
<td>Q7</td>
<td>(\text{W} { ?n1 \ \text{rdf:} \text{type} \ ?t \ \text{O}{ ?n1 \ \text{hyponymOf} \ ?n2 \ \text{O}{ ?n2 \ \text{hyponymOf} \ ?n3 \ \text{O}{ ?n3 \ \text{hyponymOf} \ ?n4 \ \text{O}{ ?n4 \ \text{hyponymOf} \ ?n5 \ \text{O}{ ?n5 \ \text{hyponymOf} \ ?n6 \ \text{O}{ ?n6 \ \text{hyponymOf} \ ?n7 }}}}}} )</td>
</tr>
<tr>
<td>Q8</td>
<td>Same as Q1, except (?b) is substituted with (301947487)</td>
</tr>
<tr>
<td>Q9</td>
<td>Same as Q2, except (?c) is substituted with (100283103)</td>
</tr>
<tr>
<td>Q10</td>
<td>Same as Q3, except all the occurrences of (?c) are substituted with (100283103)</td>
</tr>
<tr>
<td>Q11</td>
<td>Same as Q4, except all the occurrences of (?c) are substituted with (100283103)</td>
</tr>
<tr>
<td>Q12</td>
<td>Same as Q5, except all the occurrences of (?n3) are substituted with (100283103)</td>
</tr>
<tr>
<td>Q13</td>
<td>Same as Q6, except all the occurrences of (?n3) are substituted with (100283103)</td>
</tr>
<tr>
<td>Q14</td>
<td>Same as Q7, except all the occurrences of (?n2) are substituted with (100283226)</td>
</tr>
</tbody>
</table>

Table 2: WordNet test queries
0.0002 and, for most joins, are less than 0.00002. Join selectivity factor (JSF) is a factor to represent the ratio of the cardinality of a join result to the cross product of the cardinalities of the two join (twin) relations. The reason we chose queries with only joins with low selectivity factors is that the result of a join should fit into the main memory. We use a different dataset to explore the effect of join selectivity on the performance of our algorithms.

To better understand how our test SPARQL queries are translated into relational algebra and SQL and evaluated by the relational engine in our experiments, we provide the detailed description of the Q3 evaluation in the following example.

Example 5 (Q3 translation and evaluation in our experiments). Given the SPARQL query (see also Table 2)

```
01 SELECT ?a ?b ?c ?d
02 WHERE {
03   ?a rdf:type ?b .
04   OPTIONAL {
05     ?c :wordForm ?a .
```
we perform the following operations to evaluate this query in our in-memory database:

• **Query preparation.** All triple patterns in the query, \(?a \text{ rdf:type } ?b, ?c \text{ :wordForm } ?a\) and \(?c \text{ :glossaryEntry } ?d\), are evaluated, and the results are stored into the input (initial) relations, \(R_1, R_2, \text{ and } R_3\), respectively. The schema of each input relation has three attributes, denoted as \(a_1, a_2, \text{ and } a_3\), that correspond to a subject, a predicate, and an object of a matching triple. For instance, \(R_1.a_2\) always stores the value \(\text{ rdf:type }\), since all matching triples have this value as a predicate. The cardinalities of the input relations (see Table 3) are \(|R_1|=251,726\), \(|R_2|=195,802\), and \(|R_3|=111,223\). The corresponding twin relations are computed as \((R_1^l, R_1^o) = \%\{R_1\}, (R_2^l, R_2^o) = \%\{R_2\}, \text{ and } (R_3^l, R_3^o) = \%\{R_3\}\), where \(R_1^l = R_1, R_2^l = R_2, R_3^l = R_3, \text{ and } R_1^o, R_2^o, R_3^o\) are empty relations.

• **Query evaluation.** Each **OPTIONAL** clause of the query is translated into the NOJ or LOJ, such that the translation consistently uses one of the joins. The NOJ-based evaluation of Q3 is illustrated in Figure 8(a), and the LOJ-based evaluation in Figure 8(b). In the figures, the edges are annotated with the cardinalities of the corresponding (twin) relations. Note that an LOJ that corresponds to a non-nested **OPTIONAL** has no **NOT NULL** check, while any LOJ that corresponds to a nested **OPTIONAL** has such a check; a NOJ never requires this check.

For the in-RDBMS evaluation of Q3, the SPARQL query is translated into SQL. An equivalent SQL query that uses left outer joins is as follows:

```sql
01 Select * From
02 (Select t1.a as a, t1.b as b, t2.c as c From
03 t1 (Select s as a, o as b From rdf_type) t1
04 Left Outer Join
05 t2 (Select s as c, o as a From rdf_wordForm) t2
06 On (t1.a=t2.a)
07 ) t3
08 Left Outer Join
09 t4 (Select s as c, o as d From rdf_glossaryEntry) t4
10 On (t3.c=t4.c And t3.a Is Not Null)
```

Further details on the translation of SPARQL queries into relational algebra and SQL can be found in the reports of Chebotko, et al. (2006) and Cyganiak (2005). The description of the database schema generated by RDFProv in RDBMS MySQL can be found in reports of Chebotko, Fei, Lin, et al. (2007) and Chebotko, Fei, Lu, and Fotouhi (2007).

**Experiment I: Comparison of NL-NOJ and NL-LOJ**

Figure 9(a) shows the in-memory evaluation time of queries Q1-Q7 using algorithms **NL-NOJ** and **NL-LOJ**. **NL-NOJ** significantly outperformed **NL-LOJ** for all queries, except for Q1 and Q2; however, the overall performance of these algorithms on the test queries was not satisfactory. For
example, for the quite simple Q3, the \textit{NL-NOJ}-based evaluation took 13,755s, and the \textit{NL-LOJ}-based evaluation took 16,262s. These algorithms are to be used for joins with high selectivity factors as we show later in this section. The \textit{NL-NOJ} and \textit{NL-LOJ} performance for individual test queries is elaborated in the following. Query Q1 was evaluated in 34s by both algorithms. The algorithms showed similar performance for Q1 and Q2, since these queries had no nested \textit{OPTIONAL}s, and therefore \textit{NL-NOJ} had no advantage over \textit{NL-LOJ}. Queries Q3-Q7 (except perhaps Q5) showed substantial advantage of \textit{NL-NOJ} over \textit{NL-LOJ}. \textit{NL-NOJ} was slightly faster (54s difference) than \textit{NL-LOJ} for Q5, since the \textit{Cardinality} of the intermediate twin relation (see Table 3) was not significant. The \textit{NL-NOJ}-based evaluation of Q7 was roughly two times faster than the corresponding \textit{NL-LOJ}-based evaluation; the reason was in the large \textit{Cardinality} of the intermediate twin relations and relatively large number of performed joins that additionally required the \textit{NOT NULL} check for the \textit{LOJ}-based evaluation.

\textbf{Experiment II: Comparison of SM-NOJ and SM-LOJ}

Figure 9(b) shows the in-memory evaluation time of queries Q1-Q7 using algorithms \textit{SM-NOJ} and \textit{SM-LOJ}. \textit{SM-NOJ} outperformed \textit{SM-LOJ} for all queries, except for Q1 and Q2. The performance difference between \textit{SM-NOJ} and \textit{SM-LOJ} was relatively smaller than the corresponding difference between \textit{NL-NOJ} and \textit{NL-LOJ} (see Q7), because the sort-merge implementations have better lower bound than the corresponding nested-loops implementations, and the \textit{SM-NOJ} and \textit{SM-LOJ} performance is closer to the best case performance for joins with low selectivity factors.

The discussion of the algorithm performance for individual test queries is similar to the one provided in Experiment I, except that query Q1 was evaluated in 0.12s by both algorithms.
Experiment III: Comparison of SH-NOJ and SH-LOJ

Figure 9(c) shows the in-memory evaluation time of queries Q1-Q7 using algorithms SH-NOJ and SH-LOJ. Although the time difference between SH-NOJ and SH-LOJ for most test queries may seem insignificant (3%-8%), since the algorithms behave close to the linear lower bound for joins with low selectivity factors, the join processing that involves I/O operations can make this difference substantial.

The discussion of the algorithm performance for individual test queries is similar to the one provided in Experiment I, except that query Q1 was evaluated in 0.06s by both algorithms.

Experiment IV: Comparison of SH-NOJ, SM-NOJ, and NL-NOJ

Figure 9(d) shows the in-memory evaluation time of queries Q1-Q7 using algorithms SH-NOJ, SM-NOJ, and NL-NOJ. In the figure, the time axis has a logarithmic scale. For the WordNet test queries that involved only joins with low selectivity factors, SH-NOJ and SM-NOJ outperformed NL-NOJ in roughly three orders of magnitude. The SH-NOJ-based evaluation showed to be roughly twice as fast as the SM-NOJ evaluation. The observed trends are implied by the theoretical analysis of the algorithms for joins with low selectivity factors: SH-NOJ behaves close to the linear lower bound $\Omega(n)$; SM-NOJ behaves close to the lower bound $\Omega(n \log n)$; and NL-NOJ has the quadratic lower bound $\Omega(n^2)$. 

Copyright © 2008, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
Experiment V: Effect of Join Selectivity on the NOJ Algorithm Performance

To explore the effect of join selectivity on the NOJ algorithm performance, we evaluated a number of joins on artificially generated twin relations. In this experiment, we fixed the cardinalities of participating twin relations to (10,000, 0) and varied the join selectivity factor. Join selectivity factor (JSF) is a factor to represent the ratio of the cardinality of a join result to the cross product of the cardinalities of the two join (twin) relations. For the nested optional join of twin relations, we define JSF as

$$JSF = \left| (R_b, R_o) \bowtie (S_b, S_o) \right| / \left| (R_b, R_o) \times (S_b, S_o) \right|,$$

where \( |(X_b, X_o)| \) means the cardinality of twin relation \( (X_b, X_o) \) and \( |(X_b, X_o)| = |X_b| + |X_o| \).

Note that in this experiment, we did not materialize the output twin relations, because they did not fit into the main memory in case of joins with high selectivity factors. Our algorithm implementations allocated required memory for each output tuple, assigned attribute values of the tuple and deallocated the tuple memory without inserting it into the output relation.

Figure 10 shows the effect of join selectivity on the performance of SH-NOJ, SM-NOJ, and NL-NOJ. Figures 10(a) and 10(b) zoom in to the performance curves on the JSF intervals of \(0.0001 \leq JSF \leq 0.005\) and \(0.005 \leq JSF \leq 0.01\), respectively; Figure 10(c) is for the larger interval of \(0.0001 \leq JSF \leq 1.0\). The algorithm performance
curves are monotonically increasing, because the number of processed output tuples increases with larger values of JSF. 

\( SH-NOJ \) showed the best performance for \( 0.0001 \leq \text{JSF} < 0.005 \) (e.g., \( SH-NOJ \) took 0.015s, \( SM-NOJ \) took 0.031s, and \( NL-NOJ \) took 10.48s for \( \text{JSF} = 0.0001 \)). Both \( SH-NOJ \) and \( SM-NOJ \) took 0.219s for \( \text{JSF} = 0.005 \). \( SM-NOJ \) was the fastest for \( 0.01 \leq \text{JSF} < 0.8 \), and \( NL-NOJ \) was the fastest for \( 0.8 \leq \text{JSF} \leq 1.0 \). The observed phenomenon can be explained by the following facts: when the processing of output tuples is neglected, (1) \( SH-NOJ \) has the constant hashing cost however degrades from linear performance to quadratic performance with the growth of JSF; (2) \( SM-NOJ \) degrades from nonlinear \( n\log n \) performance to quadratic performance with the growth of JSF; however, at the same time, the sorting cost decreases (e.g., for \( \text{JSF} = 1.0 \), input twin relations are already sorted on the join attributes); and (3) \( NL-NOJ \) has stable quadratic performance but does not require neither hashing nor sorting.

**Experiment VI: Comparison of NL-NOJ-MySQL and NL-LOJ-MySQL**

The in-RDBMS evaluation of queries Q1-Q7 using implementations \( NL-NOJ-MySQL \) and \( NL-LOJ-MySQL \) revealed a similar pattern as for the corresponding in-memory evaluations (see Figure 9(a)). Figures 11(a) and 11(b) show the evaluation time for queries Q8-Q12 and Q13-Q14, respectively, over the WordNet dataset stored into MySQL. Similar to our previous experiments, queries with one join (Q8 and Q9) showed identical performance, since no nested \textit{OPTIONAL}s were present. \( NL-NOJ-MySQL \) was considerably faster than \( NL-LOJ-MySQL \) for Q10-Q12 and several orders of magnitude faster for Q13 and Q14. Such a huge performance difference for queries Q13 and Q14 can be naturally explained by estimating the number of operations that each implementation did to compute the result. For example, to evaluate query Q13, \( NL-NOJ-MySQL \) roughly (neglecting indexes) required 90, \( 267 \times 90, 267 \) operations for the first join (see Table 3), 1 \( \times 90, 267 + 90, 266 \) for the second one, 1 \( \times 90, 267 + 90, 266 \) for the third one, 1 \( \times 90, 267 + 90, 266 \) for the fourth one, and 1 \( \times 90, 267 + 90, 266 \) for the last one, resulting in \( \approx1 \times 90, 267^2 + 8 \times 90, 267 \) operations. On the other hand, \( NL-LOJ-MySQL \) roughly required 90, \( 267 \times 90, 267 \times 90, 267 \times 90, 267 \times 90, 267 = 90, 267^6 \) operations.

Additionally, Figure 11(c) reports the performance of \( NL-NOJ-MySQL \) and \( NL-LOJ-MySQL \) over 10 WordNet datasets stored into MySQL. Note that we did not evaluate Q13 and Q14 with \( NL-LOJ-MySQL \), since these queries were extremely slow. As before, even though the dataset became 10 times larger and joins required to use more I/O operations, the NOJ performance showed to be much better than MySQL’s LOJ performance.

**Experiment VII: Comparison of NL-NOJ-MySQL with Sesame 1.2.6 and Jena 2.5.2**

Figure 12 shows the comparison of \( NL-NOJ-MySQL \) and existing RDF store Sesame 1.2.6 (Aduna, 2008). This experiment used both systems to evaluate queries Q8-Q14 over one and 10 WordNet datasets stored into MySQL as reported in Figures 12(a) and 12(b), respectively. For all the queries, \( NL-NOJ-MySQL \) showed to be significantly faster than Sesame. Note that for the larger dataset, Sesame could not evaluate Q14 reporting an insufficient memory problem. Nevertheless, Sesame’s
optimizations showed to be quite efficient in avoiding the complexity of left outer joins that we observed in the NL-LOJ-MySQL implementation (see Experiment VI). These optimizations might be also applicable to NOJ, which we will explore in the future. An additional and very significant improvement of query response
time may be achieved if sort-merge or hash NOJ algorithms are used to evaluate the previous queries with low NOJ selectivity factors.

The comparison (not in the figures) of $NL-NOJ$-MySQL and RDF store Jena 2.5.2 (Wilkinson, et al., 2003) showed that the former was several orders of magnitude faster for all the queries. For example, while $NL-NOJ$-MySQL took less than 1s on Q12 over a single WordNet dataset, Jena required over 19,000s to evaluate the same query.

**Summary**

In the following, we summarize the results of our performance study and present our recommendations for the usage of the presented NOJ algorithms:

- The nested optional join, $(R_b, R_o) \bowtie (S_b, S_o)$, showed the performance gain over the left outer join counterpart when used for the evaluation of nested optional graph patterns in both our in-memory relational database and existing RD-BMS MySQL. The NOJ superior performance is due to the following computational improvements: (1) optional relation $R_o$ is always processed in linear time by a NOJ algorithm; and (2) NOJ does not require the NOT NULL check. These improvements showed to significantly reduce the query response time in our experiments.
- For NOJs with $JSF \leq 0.005$, algorithm $SH-NOJ$ should be used for in-memory evaluation.
- For NOJs with $JSF \geq 0.8$, algorithm $NL-NOJ$ should be used for in-memory evaluation.
- For NOJs with $0.005 < JSF < 0.8$, algorithm $SM-NOJ$ should be used for in-memory evaluation.

**RELATED WORK**

The join operation defined in the relational data model (Codd, 1970, 1972) is used to combine tuples from two or more relations based on a specified join condition. Several types of joins, such as theta-join, equi-join, natural join, semi-join, self-join, full outer join, left outer join, and right outer join, are studied in database courses (Elmasri & Navathe, 2004; Kifer, et al., 2006) and implemented in RDBMSs. We introduce a new type of join, nested optional join, whose semantics naturally support the semantics of optional patterns in well-designed graph patterns (e.g., in SPARQL) (W3C, 2008). NOJ is defined on two twin relations, where each twin relation contains a base relation and an optional relation; therefore, NOJ can be viewed as a join of four conventional relations. The result of NOJ is also a twin relation, whose base relation stores tuples that have been concatenated and whose optional relation stores tuples that have been NULL padded. These semantic and structural characteristics differentiate NOJ from any other join defined in the literature. We propose NOJ as a favorable alternative to the LOJ-based implementations for the nested optional graph pattern processing with relational databases. Note that NOJ is not a replacement of LOJ; their semantics are different, such as LOJ needs a special NOT NULL check to return similar results to the NOJ results, and this check is not part of NOJ.

The join processing in relational databases has been an important research for more than 30 years, and the related literature is abundant (Mishra & Eich, 1992). To design algorithms for NOJ, we use three classical methods for implementing joins in RDBMSs: nested-loops, sort-merge, and hash-based join methods (Elmasri &
Navathe, 2004; Kifer, et al., 2006). These methods have numerous optimizations that are out of the scope of this article. Various techniques for estimating a join selectivity factor in database systems are surveyed by Mannino, Chu, and Sager (1988).

In this article, for the SPARQL-to-SQL translation, we use the translation strategy presented in our technical report (Chebotko, et al., 2006). It is worthwhile to mention that SPARQL is not the only RDF query language that supports optional graph patterns. Other examples include SeRQL (Aduna, 2006) and RDFQL (Intellidimension, 2008). These languages can also benefit from the nested optional join.

Related literature on the SPARQL-to-SQL query translation, SPARQL query processing, and optimization includes Anyanwu, Maduko, and Sheth, 2007; Bernstein, Kiefer, and Stocker, 2007; Chebotko, Fei, Lu, and Fotouhi, 2007; Chebotko, et al., 2006; Chong, Das, Eadon, and Srinivasan, 2005; Cyganiak, 2005; Harris and Shadbolt, 2005; Harth and Decker, 2005; Hartig and Heese, 2007; Hung, Deng, and Subrahmanian, 2005; Polleres, 2007; Serfiotis, Koffina, Christophides, and Tannen, 2005; Udrea, Pugliese, and Subrahmanian, 2007; and Zemke, 2006. Harris and Shadbolt (2005) show how basic graph pattern expressions, as well as simple optional graph patterns, can be translated into relational algebra expressions. Cyganiak (2005) presents a relational algebra for SPARQL and outlines rules establishing equivalence between this algebra and SQL. Chebotko, et al. (2006) present algorithms for basic and optional graph pattern translation into SQL. The W3C semantics of SPARQL (W3C, 2008) has changed since then, which was triggered by the compositional semantics presented by Perez, et al. (2006a); Perez, Arenas, and Gutierrez (2006b). The new semantics defines the same evaluation results for the most common in practice SPARQL queries with the so-called well-designed patterns (Perez et al., 2006a), but it is different from the previously used semantics for other queries. Therefore, research results on the SPARQL-to-SQL translation described previously need to be revisited to accommodate graph patterns that are not well-designed.

One of the first SPARQL-to-SQL translations that is based on the new semantics is outlined by Zemke (2006). More recently, Chebotko, Fei, Lin, et al. (2007); Chebotko, Fei, Lu, and Fotouhi (2007) define a SPARQL-to-SQL translation algorithm for basic graph pattern queries, which is optimized to select smallest relations to query based on the type information of an instance and the statistics of the size of the relations in the database, as well as to eliminate redundancies in basic graph patterns. Furthermore, Chebotko, Lu, and Fotouhi (2007) formalize a relational algebra-based semantics of SPARQL and prove its equivalence to the mapping-based semantics of SPARQL (Perez et al., 2006a); based on this semantics, they propose the first provable semantics preserving SPARQL-to-SQL translation. Polleres (2007) contributes with the translation of SPARQL queries into Datalog, along with other contributions on the extensions of SPARQL and its semantics. Anyanwu, et al. (2007) propose an extended SPARQL query language called SPARQL2L, which supports subgraph extraction queries. Serfiotis, et al. (2005) study the containment and minimization problems of RDF query fragments using a logic framework that allows to reduce these problems into their relational equivalents. Hartig and Heese (2007) propose a SPARQL query graph model and pursue query rewriting
based on this model. Harth and Decker (2005) propose optimized index structures for RDF that can support efficient evaluation of select-project-join queries and can be implemented in a relational database. Udrea, et al. (2007) propose an in-memory index structure to store RDF graph regions defined by center nodes and their associated radii; the index helps to reduce the number of joins during SPARQL query evaluation. Weiss, Karras, and Bernstein (2008) introduce a sextuple-indexing scheme that can support efficient querying of RDF data based on six types of indexes, one for each possible ordering of a subject, predicate, and object. Bernstein, et al. (2007) propose SPARQL query optimization techniques based on triple pattern selectivity estimation and evaluate them using an in-memory SPARQL query engine. Chong, et al. (2005) introduce an SQL table function into the Oracle database to query RDF data, such that the function can be combined with SQL statements for further processing. Hung, et al. (2005) study the problem of RDF aggregate queries by extending an RDF query language with the GROUP BY clause and several aggregate functions. Several research works (Bizer & Seaborne, 2004; de Laborda & Conrad, 2006; Prud’hommeaux, 2004, 2005) focus on accessing conventional relational databases using SPARQL, which requires the SPARQL-to-SQL query translation. Finally, Guo, Qasem, Pan, and Hefflin (2007); Guo, Pan, and Hefflin (2005) define requirements for Semantic Web knowledge base systems benchmarks and propose a framework for developing such benchmarks.

Our work is complementary to the aforementioned research, as well as numerous projects on relational RDF stores, including Jena (Wilkinson, et al., 2003), Sesame (Broekstra, et al., 2002), 3store (Harris & Gibbins, 2003), KAON (Volz, et al., 2003), RStar (Ma, et al., 2004), OpenLink Virtuoso (Erling, 2001), DLDB (Pan & Hefflin, 2003), RDFSuite (Alexaki, et al., 2001), DBOWL (Narayanan, et al., 2006), PARKA (Stoffel, et al., 1997), RDFProv (Chebotko, Fei, Lin, et al., 2007), and RDFBroker (Sintek & Kiesel, 2006). While relational joins can be speeded up or even eliminated by various indexing techniques, query rewriting procedures, and database schema optimizations, they cannot be avoided completely in general case. When a join has to be computed for a nested optional graph pattern, a choice between NOJ or LOJ can make a difference. Additionally, supplemental index structures that reduce the size of to-be-joined relations can be applied in conjunction with NOJ. Therefore, NOJ can be beneficial to many existing relational RDF stores.

CONCLUSION AND FUTURE WORK

To support efficient processing of RDF queries with well-designed graph patterns and nested optional patterns in RDBMSs, we proposed a novel relational operator—nested optional join. We illustrated that such RDF queries can be translated into relational algebra using either left outer join (LOJ) or nested optional join (NOJ). The computational advantage of NOJ over the currently used LOJ-based implementations comes from the two superior characteristics of NOJ: (1) NOJ allows the processing of the tuples that are guaranteed to be NULL padded very efficiently (in linear time); and (2) NOJ does not require the NOT NULL check to return correct results. In addition, (3) NOJ can significantly simplify the translation of RDF queries with
well-designed graph patterns into relational algebra. To facilitate the implementation of NOJ in relational databases, we designed three efficient algorithms: (1) nested-loops NOJ algorithm, NL-NOJ, (2) sort-merge NOJ algorithm, SM-NOJ, and (3) simple hash NOJ algorithm, SH-NOJ. Based on a real-life RDF dataset, we demonstrated the efficiency of our algorithms by comparing them with the corresponding left outer join implementations using our in-memory database and popular open-source RDBMS MySQL. The experiments showed that NOJ is a favorable alternative to the LOJ-based evaluation of nested optional patterns in well-designed graph patterns. Moreover, we conducted a preliminary performance study of the NOJ algorithm behavior with respect to varying join selectivity factor (JSF). This study showed that for NOJs with JSF ≤ 0.005, SH-NOJ should be used for in-memory evaluation; for NOJs with JSF ≥ 0.8, NL-NOJ should be used for in-memory evaluation; and for NOJs with 0.005 < JSF < 0.8, SM-NOJ should be used for in-memory evaluation. Finally, our last experiment showed that our NOJ-based implementation outperformed existing RDF stores Sesame and Jena.

In the future, we would like to explore NOJ performance in an RDBMS using sort-merge and hash join algorithms, conduct a performance study on larger and diverse datasets, seek for additional optimization opportunities for NOJ and its implementations, and experiment with NOJ implementations in a column-oriented database (Abadi, et al., 2007; Sidirourgos, Goncalves, Kersten, Nes, & Manegold, 2008). We would also like to compare top-down (Chebotko, et al., 2006; Cyganiak, 2005) (also known as depth-first) (Perez, et al., 2006a), and bottom-up (Chebotko, Lu, & Fotouhi, 2007; Perez, et al., 2006a) approaches to evaluation of nested optional patterns in well-designed graph patterns. Last but not least, we are interested in exploring NOJ-aware reordering techniques for query optimization (Glalindo-Legaria & Rosenthal, 1997; Roa, Lindsay, Lohman, Pirahesh, & Simmen, 2001).


ence/db/default.rsp


ENDNOTES

1 Note that an attribute that serves as an indicator whether the parent OPTIONAL clause has succeeded should be carefully chosen as discussed by Chebotko, et al. (2006). In a nutshell, such an attribute may not be bound in any clause that precedes the parent OPTIONAL; otherwise, the NOT NULL check may succeed, even if the parent OPTIONAL fails.
Artem Chebotko is an assistant professor in the Department of Computer Science at University of Texas - Pan American. He received his PhD in computer science from Wayne State University in 2008, MA in computer science from Wayne State University in 2005, MS in management information systems from Ukraine State Maritime Technical University in 2003, and B.S. in Computer Science from Ukraine State Maritime Technical University in 2001. His research interests include scientific workflow provenance metadata management, semantic web data management, and database systems. Dr. Chebotko has published a number of papers in refereed journals and conference proceedings and currently serves as a program committee member of several international conferences and workshops on scientific workflows and semantic web. He is a member of ACM and IEEE.

Shiyong Lu is currently an assistant professor in the Department of Computer Science at Wayne State University and the director of the Scientific Workflow Research Laboratory (the SWR Lab). Dr. Lu received his PhD from State University of New York at Stony Brook in 2002, ME from the Institute of Computing Technology of Chinese Academy of Sciences at Beijing in 1996, and BE from the University of Science and Technology of China at Hefei in 1993. His research interests include scientific workflows and Semantic Web. He has published over sixty refereed international journal and conference papers in the above areas. Dr. Lu is the founder and currently the chair of the IEEE International Workshop on Scientific Workflows (2007~2009), an editorial board member for International Journal of Semantic Web and Information Systems and International Journal of Medical Information Systems and Informatics. He also serves as a program committee member for several top-tier IEEE conferences. He is a member of IEEE.

Mustafa Atay received his PhD in computer science from Wayne State University in 2006. He is currently an assistant professor in the Department of Computer Science at Winston-Salem State University. His research interests include XML data management, database systems, Semantic Web, data integration and information retrieval. He has published several refereed international journal and conference papers. He has served as a program committee member of conferences on Web technologies and information systems. He is a member of ACM.

Farshad Fotouhi received his PhD in computer science from Michigan State University in 1988. He joined the faculty of computer science at Wayne State University in August 1988 where he is currently Professor and Chair of the department. His major areas of research include XML databases, semantic web, multimedia systems, and query optimization. He has published over 100 papers in refereed journals and conference proceedings, served as program committee member of various database related conferences. Dr. Fotouhi is on the editorial boards of the IEEE Multimedia Magazine and The International Journal on Semantic Web and Information Systems.