

Extending Relational Databases with a Nested Optional Join for Efficient Semantic Web Query Processing

(Technical Report TR-DB-112006-CALF, November 2006. Working Draft.)

Artem Chebotko, Mustafa Atay, Shiyong Lu 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. Increasing amount of RDF metadata on the Web highlights the need for its efficient and effective management. In this light, numerous researchers have proposed to use RDBMSs to store and query RDF annotations using the SQL and SPARQL query languages. The first few attempts in SPARQL-to-SQL translation revealed non-trivial challenges related to correctness and efficiency of such translation in the presence of nested optional graph patterns. Instead of trying to reconcile semantically different languages, SPARQL and SQL, 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 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*. In our empirical study based on the real-life RDF metadata, we verify the efficiency of our algorithms and compare them with the corresponding left outer join implementations. The experimental results are very promising; the NOJ is a favorable alternative to the LOJ-based evaluation of nested optional patterns in the Semantic Web query processing with RDBMSs.

1 Introduction

The Semantic Web [7] 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 metadata is represented in Resource Description Framework (RDF) [1], the standard language for annotating resources on the Web, and queried using the SPARQL [4] query language for RDF that has been recently proposed by the World Wide Web Consortium. RDF metadata is a collection of statements, called *triples*, of the form $\langle s, p, o \rangle$, 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 metadata on the Web highlights the need for its efficient and effective management. In this light, numerous researchers [13, 9, 26, 25, 18, 28, 8, 27, 24] have proposed to use RDBMSs to store and query RDF metadata 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. The first few attempts [9, 13] in the SPARQL-to-SQL translation, although successful, revealed serious difficulties related to correctness and efficiency of such translation in the presence of nested optional graph patterns. The challenges of the SPARQL query processing in the presence of nested **OPTIONAL** patterns include:

- *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 OPTIONAL patterns.* 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 nested OPTIONAL patterns.* Before a nested OPTIONAL clause is evaluated, all containing OPTIONAL clauses must have succeeded.

In existing 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 patterns is captured by the 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 patterns is preserved by the NOT NULL check in the LOJ condition for one of the attributes/variables that correspond to the parent of a nested OPTIONAL clause.

The first two semantics are successfully addressed in the literature (e.g., [9, 13, 18, 26]), however the nested OPTIONAL semantics is showed to be more intriguing. Two recent works that consider the SPARQL-to-SQL translation in the presence of nested OPTIONALS are [13] and [9]. 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 ... does not fully reproduce SPARQL semantics” [13]. Our algorithm SPARQLtoSQL presented in [9] is the first semantics preserving solution to the SPARQL-to-SQL translation problem in the presence of arbitrary complex optional graph patterns. In the following, we present our running example to illustrate the semantics preserving translation of a SPARQL query with nested OPTIONALS into the relational algebra, in which the LOJ is used for implementing nested optional graph patterns; the example also motivates the introduction of a new relational operator for a more efficient implementation.

Example 1.1 (Sample SPARQL query and its relational equivalent)

Consider the RDF graph presented in Figure 1(a). The graph describes the academic tree of Arthur Bernstein using the *academicChildOf* relation. The RDF schema defines two concepts/classes (*Professor* and *GradStudent*) and two relations/properties (*academicChildOf* and *collaboratesWith*). The domain of *academicChildOf* includes *Professor* and *GradStudent*, and the range includes *Professor*. The domain and range of *collaboratesWith* include *GradStudent*. Additionally, four instances of *Professor*, two instances of *GradStudent* and relations between these instances are defined as shown in the figure.

We design a SPARQL query that retrieves (1) every graduate student in the RDF graph; (2) the student’s advisor if this information is available; and (3) the student’s collaborators if this information is available and if the student’s advisor has been successfully retrieved in the previous step. The SPARQL representation of the query is as follows.

```

01 SELECT ?stud1 ?prof ?stud2
02 WHERE {
03   ?stud1 rdf:type :GradStudent .      /* R1(stud1) */
04   OPTIONAL {
05     ?stud1 :academicChildOf ?prof .    /* R2(stud1,prof) */
06     OPTIONAL {
07       ?stud1 :collaboratesWith ?stud2 /* R3(stud1,stud2) */
08     } } }

```

The query has three variables: `?stud1` for the first student, `?prof` for the advisor, and `?stud2` for the student collaborator. There are two OPTIONAL clauses, where the outermost one is the nested OPTIONAL clause.

Based on our translation strategy in [9], we translate the SPARQL query into a relational query. Let matching triples for the triple patterns `?stud1 rdf:type :GradStudent`, `?stud1 :academicChildOf ?prof` and `?stud1 :collaboratesWith ?stud2` 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 SPARQL query above. Then the equivalent relational algebra representation is

$$R_4 = \Pi_{R_1.stud1, R_2.prof} (R_1 \bowtie_{R_1.stud1=R_2.stud1} R_2),
R_{res} = \Pi_{R_4.stud1, R_4.prof, R_3.stud2} (R_4 \bowtie_{R_4.stud1=R_3.stud1 \wedge R_4.prof \text{ IS NOT NULL } R_3}).$$

Each `OPTIONAL` clause corresponds to the left outer join, the shared variable `?stud1` participates in the join conditions, and the nested `OPTIONAL` implements the `NOT NULL` check on the `prof` 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 the ease of presentation. \diamond

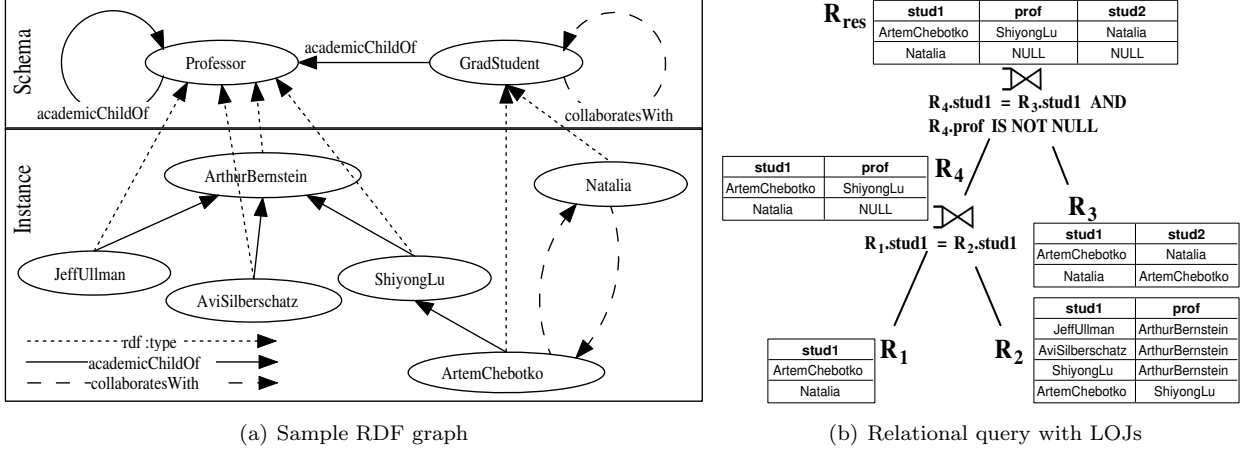


Figure 1: Sample RDF graph and a relational query over the graph

The running example also motivates our research. The following is our insight on 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` pattern, we know that the `NULL` padded tuple $(Natalia, \text{NULL})$ will also be `NULL` padded in the second LOJ. After all, there is no need for this tuple to participate in the second LOJ condition. (2) The `NOT NULL` check in the nested LOJ looks like a patch that fixes the inconsistency in the SPARQL and SQL semantics. There is no need to apply this check to the successful match in the tuple $(ArtemChebotko, ShiyongLu)$.

In this paper, instead of trying to reconcile semantically different languages, SPARQL and SQL, we propose to extend relational databases with an innovative operator that imitates the nested optional pattern semantics of SPARQL to enable efficient processing of nested optional patterns in RDBMSs. The main contributions of our work include:

- 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 graph patterns. The computational advantage of the NOJ over the currently used LOJ-based implementations comes from the two superior characteristics of the NOJ: (1) the NOJ allows the processing of the tuples that are guaranteed to be `NULL` padded very efficiently, in linear time and (2) the NOJ does not require the `NOT NULL` check to return correct results.
- 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*.
- In our empirical study based on the real-life RDF metadata, we verify the efficiency of our algorithms and compare them with the corresponding left outer join implementations. The experimental results are very promising; the NOJ is a favorable alternative to the LOJ-based evaluation of nested optional patterns in the Semantic Web query processing with RDBMSs.
- 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.

Organization. The rest of the paper is organized as follows. In Section 2, we present our NOJ operator, highlight its advantages and explore its properties. We design three algorithms to implement the NOJ in relational databases in Section 3 and report the results of our performance study in Section 4. In Section 5, we discuss related work. Finally, we provide our conclusions and future work in Section 6.

Key words: Nested Optional Join, Relational Join, Relational Operator, RDBMS, SPARQL, RDF, Semantic Web, Query Processing.

2 Nested Optional Join Operator

In this section, we present our nested optional join operator that is to be used to evaluate nested **OPTIONAL** patterns in relational databases, highlight its advantages over the left outer join and explore some of its properties.

We begin with the definition of the special kind of a relation, *twin relation*, that is used to specify input and output of the NOJ.

Definition 2.1 (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 $\xi(R_b, R_o)$ is called *base relation*, and R_o with the schema $\xi(R_b, R_o)$ is called *optional relation*. A distinguished tuple $n_{\xi(R_b, R_o)}$ is defined as a tuple of $\xi(R_b, R_o)$ in which each attribute takes a NULL value. \diamond

A base relation is intended to store tuples that have a potential to satisfy a join condition if used in a nested optional join. An optional relation is intended to store tuples that are guaranteed to fail a join condition if used in a nested optional join. To fill the gap between the twin relation and the relational algebra, we provide a simple *conversion operator*, \rightarrow , that can convert a twin relation to a conventional relation and back:

- $(R_b, R_o) \rightarrow R$, a twin relation (R_b, R_o) can be converted into a conventional relation R , such that $R = R_b \cup R_o$.
- $R \rightarrow (R_b, R_o)$, a conventional relation R can be represented as a twin relation (R_b, R_o) , such that $R_b = R$ and R_o is an empty relation.

The conversion operator should be used carefully due to its following property: if $(R_b, R_o) \rightarrow R \rightarrow (R'_b, R'_o)$, then $R_b \neq R'_b$ and $R_o \neq R'_o$, unless R_o is an empty relation. Therefore, unnecessary conversions should be avoided. For this purpose, we 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 paper; π and σ 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 2.2 (Nested Optional Join) A nested optional join of two twin relations, denoted as $\exists\bowtie$, yields a twin relation, such that $(R_b, R_o) \exists\bowtie_{r(a)=s(b)} (S_b, S_o) = (Q_b, Q_o)$, where $Q_b = \{t | t = rs \wedge r \in R_b \wedge (s \in S_b \vee s \in S_o) \wedge t(a) = t(b)\}$ and $Q_o = \{t | t = rn \wedge (r \in R_o \vee (r \in R_b \wedge \neg \exists s[(s \in S_b \vee s \in S_o) \wedge r(a) = s(b)]))\}$, where $r(a) = s(b)$ is a join predicate, $r(a) \subseteq \xi(R_b, R_o)$ and $s(b) \subseteq \xi(S_b, S_o)$ 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 enforces, or r must be a tuple in R_b and there must no 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 the 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 the NOJ – no need for the NOT NULL check – is discussed in the following example that describes the translation of our sample SPARQL query into the relational algebra with our extensions.

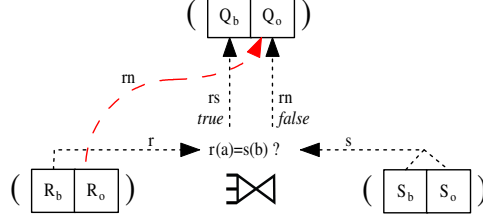


Figure 2: Nested optional join

Example 2.3 (Evaluation of the sample SPARQL query using NOJs)

We use the same RDF graph as presented in Figure 1(a) and the SPARQL query described in Example 1.1. The translation strategy is also similar to [9], but we use the NOJ instead of the LOJ. Let matching triples for the triple patterns `?stud1 rdf:type :GradStudent`, `?stud1 :academicChildOf ?prof` and `?stud1 :collaboratesWith ?stud2` are retrieved into relations R_1 , R_2 and R_3 , respectively. Then the equivalent relational algebra representation is

$$\begin{aligned}
 R_1 &\rightarrow (R_b^1, R_o^1), R_2 \rightarrow (R_b^2, R_o^2), R_3 \rightarrow (R_b^3, R_o^3), \\
 (R_b^4, R_o^4) &= \Pi_{(R_b^1, R_o^1).stud1, (R_b^2, R_o^2).prof}((R_b^1, R_o^1) \bowtie_{(R_b^1, R_o^1).stud1=(R_b^2, R_o^2).stud1} (R_b^2, R_o^2)), \\
 (R_b^{res}, R_o^{res}) &= \Pi_{(R_b^4, R_o^4).stud1, (R_b^3, R_o^3).stud2}((R_b^4, R_o^4) \bowtie_{(R_b^4, R_o^4).stud1=(R_b^3, R_o^3).stud1} (R_b^3, R_o^3)).
 \end{aligned}$$

The graphical representation of the relational query is shown in Figure 3; the conversion and projection operators are not shown for the 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 and, thus, are padded with NULL values are stored into the optional relation R_o^4 ; the tuples of R_o^4 bypass the second join condition and are copied directly to R_o^{res} with additional NULL-padding. \diamond

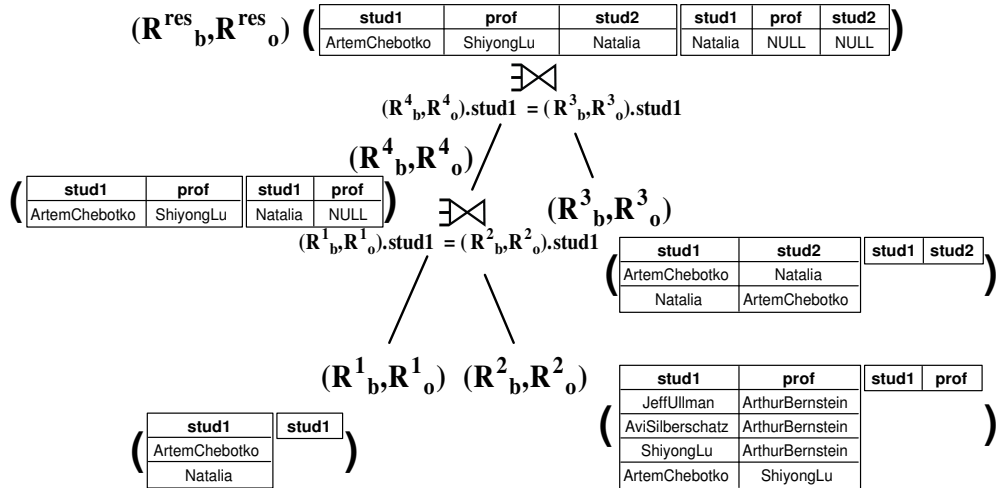


Figure 3: Nested optional join based evaluation of the SPARQL query in Example 1.1

Therefore, the NOJ is superior to the LOJ when we apply them to translate SPARQL nested **OPTIONAL** clauses to relational queries. The two main advantages of the NOJ are (1) the tuples that are guaranteed to be **NULL** padded can be processed very efficiently, in linear time, and (2) the **NOT NULL** check is eliminated. Our performance study showed that these advantages bring substantial speedup to the query evaluation. Additionally, we can identify two other advantages of the NOJ: (3) a SPARQL-to-SQL translation does not need to choose an attribute to perform the **NOT NULL** check¹ and (4) in some cases as described in [9], when such an attribute can not be chosen from the available ones, the introduction of a new variable and even a new triple pattern in a SPARQL query may be required, which increases the complexity of the query; again, this processing overhead is eliminated for the NOJ.

We explore a few important properties of the nested optional join that are similar to the properties of the LOJ:

1. *Non-commutativity*

$$(R_b, R_o) \bowtie_p (S_b, S_o) \not\iff (S_b, S_o) \bowtie_p (R_b, R_o)$$

Proof: Let r_b, r_o, s_b and s_o denote tuples from R_b, R_o, S_b and S_o , respectively, $n_R = n_{\xi(R_b, R_o)}$ and $n_S = n_{\xi(S_b, S_o)}$.

Let $(R_b, R_o) \bowtie_p (S_b, S_o) = (Q_b^I, Q_o^I)$, then $Q_b^I = \{(r_b s_b \wedge p) \vee (r_b s_o \wedge p)\}$ and $Q_o^I = \{r_o n_S \vee (r_b n_S \wedge \neg \exists s_b, s_o[p])\}$.

Let $(S_b, S_o) \bowtie_p (R_b, R_o) = (Q_b^{II}, Q_o^{II})$, then $Q_b^{II} = \{(s_b r_b \wedge p) \vee (s_b r_o \wedge p)\}$ and $Q_o^{II} = \{s_o n_R \vee (s_b n_R \wedge \neg \exists r_b, r_o[p])\}$.

Therefore, $Q_b^I \neq Q_b^{II}$, $Q_o^I \neq Q_o^{II}$ and $(Q_b^I, Q_o^I) \neq (Q_b^{II}, Q_o^{II})$. \square

2. *Non-associativity*

$$((R_b, R_o) \bowtie_{p_1} (S_b, S_o)) \bowtie_{p_2} (T_b, T_o) \not\iff (R_b, R_o) \bowtie_{p_1} ((S_b, S_o) \bowtie_{p_2} (T_b, T_o))$$

Proof: Let r_b, r_o, s_b, s_o, t_b and t_o denote tuples from R_b, R_o, S_b, S_o, T_b and T_o , respectively, $n_S = n_{\xi(S_b, S_o)}$ and $n_T = n_{\xi(T_b, T_o)}$.

Let $(R_b, R_o) \bowtie_{p_1} (S_b, S_o) = (Q_b^I, Q_o^I)$, then $Q_b^I = \{(r_b s_b \wedge p_1) \vee (r_b s_o \wedge p_1)\}$ and $Q_o^I = \{r_o n_S \vee (r_b n_S \wedge \neg \exists s_b, s_o[p_1])\}$.

Let $(Q_b^I, Q_o^I) \bowtie_{p_2} (T_b, T_o) = (Q_b^{II}, Q_o^{II})$, then $Q_b^{II} = \{(r_b s_b t_b \wedge p_1 \wedge p_2) \vee (r_b s_b t_o \wedge p_1 \wedge p_2) \vee (r_b s_o t_b \wedge p_1 \wedge p_2) \vee (r_b s_o t_o \wedge p_1 \wedge p_2)\}$ and $Q_o^{II} = \{r_o n_S n_T \vee (r_b n_S n_T \wedge \neg \exists s_b, s_o[p_1]) \vee (r_b s_b n_T \wedge p_1 \wedge \neg \exists t_b, t_o[p_2]) \vee (r_b s_o n_T \wedge p_1 \wedge \neg \exists t_b, t_o[p_2])\}$.

Let $(S_b, S_o) \bowtie_{p_2} (T_b, T_o) = (Q_b^{III}, Q_o^{III})$, then $Q_b^{III} = \{(s_b t_b \wedge p_2) \vee (s_b t_o \wedge p_2)\}$ and $Q_o^{III} = \{s_o n_T \vee (s_b n_T \wedge \neg \exists t_b, t_o[p_2])\}$.

Let $(R_b, R_o) \bowtie_{p_2} (Q_b^{III}, Q_o^{III}) = (Q_b^{IV}, Q_o^{IV})$, then $Q_b^{IV} = \{(r_b s_b t_b \wedge p_1 \wedge p_2) \vee (r_b s_b t_o \wedge p_1 \wedge p_2) \vee (r_b s_o n_T \wedge p_1) \vee (r_b s_b n_T \wedge \neg \exists t_b, t_o[p_2] \wedge p_1)\}$ and $Q_o^{IV} = \{r_o n_S n_T \vee (r_b n_S n_T \wedge \neg \exists s_b, s_o[p_1])\}$.

It is easy to show that $(Q_b^{II}, Q_o^{II}) \neq (Q_b^{IV}, Q_o^{IV})$. \square

3. *Left-associativity (SPARQL-to-SQL translation property)*

$$(R_b, R_o) \bowtie_{p_1} (S_b, S_o) \bowtie_{p_2} (T_b, T_o) \iff ((R_b, R_o) \bowtie_{p_1} (S_b, S_o)) \bowtie_{p_2} (T_b, T_o)$$

Strictly speaking, this is not the NOJ property, but the SPARQL-to-SQL translation property, which is introduced in the SPARQL specification for **OPTIONAL** clauses. Since the NOJ was designed for such a translation, we highlight this property here.

Previously, we defined the NOJ through the tuple calculus, but it is also possible to express the NOJ result relations using standard operators of the relational algebra: $Q_b = R_b \bowtie_{r(a)=s(b)} (S_b \cup S_o)$ and $Q_o = R_o \cup [(R_b \bowtie_{r(a)=s(b)} (S_b \cup S_o)) - (R_b \bowtie_{r(a)=s(b)} (S_b \cup S_o))]$. However, it should be evident that this direct translation will be inefficient if implemented. Therefore, in the next section, we design our own algorithms to implement the NOJ.

3 Nested Optional Join Algorithms

We design three algorithms, NL-NOJ, SM-NOJ and SH-NOJ, to evaluate nested optional joins in a relational database. Our algorithms employ the classic methods used to implement relational joins: nested-loops, sort-merge and hash-based join methods.

¹Note that the attribute that serves as an indicator whether the parent **OPTIONAL** clause has succeeded should be carefully chosen as discussed in [9]. 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.

3.1 Nested-loops nested optional join algorithm

The simplest algorithm to perform the NOJ operation is the nested-loops NOJ algorithm, denoted as NL-NOJ. The algorithm (see Figure 4) is self-descriptive, and thus we only clarify some important details. Note that for efficiency, the inner loop in line 07 should iterate over the tuples of a (twin) relation with higher cardinality. This remark is only valid when I/O operations are involved, and can be ignored for in-memory join processing. In the figure, we assume that (S_b, S_o) has more tuples than R_b . Also, note that the tuples of R_o are processed in linear time in lines 17-19.

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 in [23]. 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 [20, 16] and “rocking” the inner relation optimization [21] that reduce the number of I/O operations.

```

01 Algorithm: NL-NOJ
02 Input: twin relations  $(R_b, R_o)$  and  $(S_b, S_o)$ 
03 Output: twin relation  $(Q_b, Q_o) = (R_b, R_o) \bowtie_{r(a)=s(b)} (S_b, S_o)$ 
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  $Q_b$ 
10        $pad = false$ 
11     End If
12   End For
13   If  $pad$  then
14     place tuple  $rn_{\xi(S_b, S_o)}$  in relation  $Q_o$ 
15   End If
16 End For
17 For each tuple  $r \in R_o$  do
18   place tuple  $rn_{\xi(S_b, S_o)}$  in relation  $Q_o$ 
19 End For
20 Return  $(Q_b, Q_o)$ 
21 End Algorithm

```

Figure 4: Algorithm NL-NOJ

3.2 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 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

- *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_b , 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 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).

The comprehensive analysis of the performance and applicability of the sort-merge join method is presented in [23].

```

01 Algorithm: SM-NOJ
02 Input: twin relations  $(R_b, R_o)$  and  $(S_b, S_o)$ 
03 Output: twin relation  $(Q_b, Q_o) = (R_b, R_o) \bowtie_{r(a)=s(b)} (S_b, S_o)$ 
04 Begin
05   Sort  $R_b$  on  $r(a)$ 
06   Let  $S = S_b \cup S_o$ 
07   Sort  $S$  on  $s(b)$ 
08    $r =$  first tuple of  $R_b$ 
09    $s =$  first tuple of  $S$ 
10   While  $r \neq EOF$  do
11     If  $s = EOF$  then
12       place tuple  $rn_{\xi(S_b, S_o)}$  in relation  $Q_o$ 
13        $r =$  next tuple after  $r$  of  $R_b$ 
14     Else
15       While  $r \neq EOF$  and  $r(a) < s(b)$  do
16         place tuple  $rn_{\xi(S_b, S_o)}$  in relation  $Q_o$ 
17          $r =$  next tuple after  $r$  of  $R_b$ 
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  $r(a) = s(b)$  do
26           place tuple  $rs$  in relation  $Q_b$ 
27            $s =$  next tuple after  $s$  of  $S$ 
28         End While
29         If  $r$  and next tuple after  $r$  of  $R_b$  have the same values of  $r(a)$  then
30            $back = true$ 
31         End If
32          $r =$  next tuple after  $r$  of  $R_b$ 
33       End If
34       If  $back$  then  $s = backtrack$  End If
35     End If
36   End While
37   For each tuple  $r \in R_o$  do
38     place tuple  $rn_{\xi(S_b, S_o)}$  in relation  $Q_o$ 
39   End For
40   Return  $(Q_b, Q_o)$ 
41 End Algorithm

```

Figure 5: Algorithm SM-NOJ

3.3 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 non-empty 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) [23].

The results of our complexity and applicability analysis are

- *SH-NOJ complexity*: $\Omega(|R_b| + |R_o| + |S_b| + |S_o|)$, $O(|R_b| \times (|S_b| + |S_o|) + |R_o|)$, 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, the slower SH-NOJ. In the worse 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 in [23]. The simple hash join method [15] that is used to implement SH-NOJ is not the only hash-based join method that is found in the join processing literature. Other methods, such as simple hash-partitioned join, GRACE hash join, hybrid hash join and hashed loops join [14, 22, 17], can be also adapted to perform a hash-based NOJ.

```

01 Algorithm: SH-NOJ
02 Input: twin relations  $(R_b, R_o)$  and  $(S_b, S_o)$ 
03 Output: twin relation  $(Q_b, Q_o) = (R_b, R_o) \bowtie_{r(a)=s(b)} (S_b, S_o)$ 
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  $r \in R_b$  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  $Q_b$ 
14      $pad = false$ 
15   End For
16   If  $pad$  then
17     place tuple  $rn_{\xi(S_b, S_o)}$  in relation  $Q_o$ 
18   End If
19   End For
20   For each tuple  $r \in R_o$  do
21     place tuple  $rn_{\xi(S_b, S_o)}$  in relation  $Q_o$ 
22   End For
23   Return  $(Q_b, Q_o)$ 
24 End Algorithm

```

Figure 6: Algorithm SH-NOJ

4 Performance Study

This section reports the performance experiments conducted using the NOJ algorithms and an in-memory relational database. The performance of the NOJ algorithms is compared with the performance of the corresponding LOJ-based implementations. The behavior of the NOJ algorithms with respect to the NOJ selectivity factor is explored.

4.1 Experimental Setup

We implemented in-memory representations of a relation and a twin relation, such that the relation is represented by the double-linked list of tuples, where each tuple corresponds to an array of pointers to attribute data values, and the twin relation is represented by pointers to two conventional relations. All 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++ using MS Visual C++ 6.0. 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, e.g., [23]).

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

4.2 Data set

We conducted the experiments using the OWL representation of WordNet [6] (version: 1.2; author: Claudia Ciorascu), 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 `owl:Class`, and each linguistic relation is modeled as an `owl:ObjectProperty`, `owl:DatatypeProperty`, `owl:TransitiveProperty`, `owl:ObjectProperty` or `owl:SymmetricProperty`. The simplified WordNet ontology is illustrated in Figure 7. The figure does not include some classes (e.g., `wn:Nouns_and_Verbs`) and properties (e.g., `wn:mMeronym`) that are not essential for the understanding of the data set and the experiments.

The relevant statistics for the WordNet data set is shown in Table 1. For example, *WordNet.OWL* contains 251,726 triples involving `rdf:type` as the predicate, and 140,470 of them have `wn:WordObject` as the object.

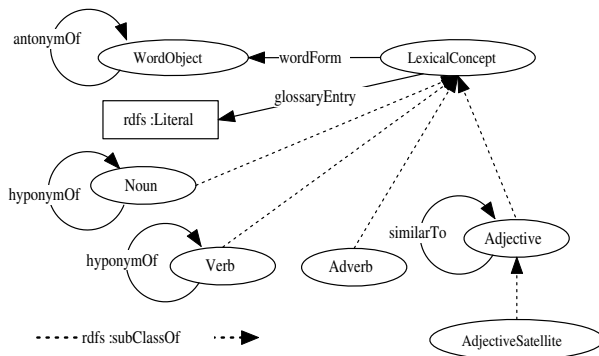


Figure 7: Simplified WordNet ontology

Table 1: Property and resource statistics of WordNet

Property	Count	Resource	Count
<code>type</code>	251,726	<code>WordObject</code>	140,470
<code>wordForm</code>	195,802	<code>Noun</code>	75,804
<code>glossaryEntry</code>	111,223	<code>Verb</code>	13,214
<code>hyponymOf</code>	90,267	<code>AdjectiveSatellite</code>	11,231
<code>similarTo</code>	22,494	<code>Adjective</code>	7,345
<code>antonymOf</code>	7,115	<code>Adverb</code>	3,629
<i>Others</i>	36,225	<i>Others</i>	33
Total	714,852	Total	251,726

4.3 Test Queries

We chose nine SPARQL [4] queries to evaluate in our experiments based on the following criterion: (1) most queries should have nested `OPTIONAL` clauses; (2) the input, intermediate and output (twin) relations involved in the query evaluation should fit into the main memory; and (3) 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 is omitted for brevity, and the projection includes all attributes of the join participating (twin) relations. 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 clause 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 OPTIONALS in the queries. Queries Q2-Q6 and similarly Q7-Q9 are related and can be derived from each other. Q9 stands out from Q8 and Q7 in considerably larger values of Cardinality_o in the intermediate twin relations.

An important characteristic of the test queries is that they only involve joins, whose selectivity factors are less than 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 why 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 data set to explore the effect of join selectivity on the performance of our algorithms.

Table 2: Test queries

#	Query
Q1	<code>W{?a rdf:type :Adjective O{?a :similarTo ?b}}</code>
Q2	<code>W{?a rdf:type ?b O{?c :wordForm ?a}}</code>
Q3	<code>W{?a rdf:type ?b O{?c :wordForm ?a O{?c :glossaryEntry ?d}}</code>
Q4	<code>W{?a rdf:type ?b O{?c :wordForm ?a O{?c :glossaryEntry ?d O{?c :hyponymOf ?e}}}}</code>
Q5	<code>W{?a rdf:type ?b O{?c :wordForm ?a O{?c :glossaryEntry ?d O{?c :hyponymOf ?e O{?a antonymOf ?f}}}}}}</code>
Q6	<code>W{?a rdf:type ?b O{?c :wordForm ?a O{?c :glossaryEntry ?d O{?c :hyponymOf ?e O{?a antonymOf ?f O{?c rdf:type ?g}}}}}}}}</code>
Q7	<code>W{?n1 :hyponymOf ?n2 O{?n2 :hyponymOf ?n3 O{?n3 :hyponymOf ?n4}}</code>
Q8	<code>W{?n1 :hyponymOf ?n2 O{?n2 :hyponymOf ?n3 O{?n3 :hyponymOf ?n4 O{?n4 :hyponymOf ?n5 O{?n5 :hyponymOf ?n6 O{?n6 :hyponymOf ?n7}}}}}}}}</code>
Q9	<code>W{?n1 rdf:type ?t O{?n1 :hyponymOf ?n2 O{?n2 :hyponymOf ?n3 O{?n3 :hyponymOf ?n4 O{?n4 :hyponymOf ?n5 O{?n5 :hyponymOf ?n6 O{?n6 :hyponymOf ?n7}}}}}}}}}}</code>

Table 3: Characteristics of the test queries

Query #	# Joins	Twin relation cardinality: (Cardinality _b ; Cardinality _o)		
		Input	Intermediate	Output
Q1	1	(7,345; 0) (22,494; 0)	N/A	(11,249; 4,653)
Q2	1	(251,726; 0) (195,802; 0)	N/A	(195,803; 111,255)
Q3	2	(251,726; 0) (195,802; 0) (111,223; 0)	(195,803; 111,255)	(195,803; 111,255)
Q4	3	(251,726; 0) (195,802; 0) (111,223; 0) (90,267; 0)	(195,803; 111,255) (195,803; 111,255)	(161,247; 149,656)
Q5	4	(251,726; 0) (195,802; 0) (111,223; 0) (90,267; 0) (7,115; 0)	(195,803; 111,255) (195,803; 111,255) (161,247; 149,656)	(10,682; 302,014)
Q6	5	(251,726; 0) (195,802; 0) (111,223; 0) (90,267; 0) (7,115; 0) (251,726; 0)	(195,803; 111,255) (195,803; 111,255) (161,247; 149,656) (10,682; 302,014)	(10,682; 302,014)
Q7	2	(90,267; 0) (90,267; 0) (90,267; 0)	(89,220; 3,416)	(88,213; 8,591)
Q8	5	(90,267; 0) (90,267; 0) (90,267; 0) (90,267; 0) (90,267; 0)	(89,220; 3,416) (88,213; 8,591) (86,323; 15,872) (81,263; 27,274)	(67,788; 45,800)
Q9	6	(251,726; 0) (90,267; 0) (90,267; 0) (90,267; 0) (90,267; 0) (90,267; 0)	(90,267; 163,343) (89,220; 166,759) (88,213; 171,934) (86,323; 179,215) (81,263; 190,617)	(67,788; 209,143)

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

Example 4.1 (Q3 translation and evaluation in our experiments) Given the SPARQL query `W{?a rdf:type ?b O{?c :wordForm ?a O{?c :glossaryEntry ?d}}` (see Table 2), we perform the following operations:

- *Query preparation.* All triple patterns in the query, `?a rdf:type ?b`, `?c :wordForm ?a` and `?c :glossaryEntry ?d`, are evaluated and the results are stored into the input (initial) relations, R_1 , R_2 and R_3 , respectively. The relational schema of each input relation has three attributes, denoted as

a_1 , a_2 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 `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 $R_1 \rightarrow (R_b^1, R_o^1)$, $R_2 \rightarrow (R_b^2, R_o^2)$ and $R_3 \rightarrow (R_b^3, R_o^3)$, where $R_b^1 = R_1$, $R_b^2 = R_2$, $R_b^3 = R_3$ and R_o^1, R_o^2, R_o^3 are empty relations. Note that the query preparation time is not measured in our experiments.

- *Query evaluation.* Each **OPTIONAL** clause of the query is translated into the **NOJ** or the **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 a **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.

◇

Further details on the translation of SPARQL queries into relational algebra and SQL are available in [9].

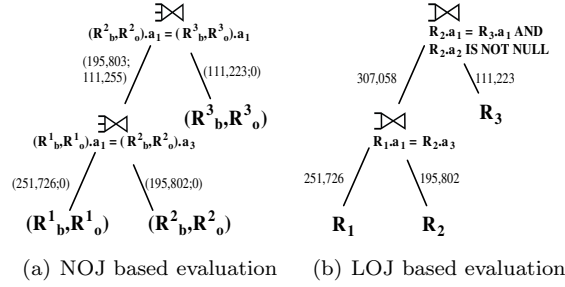


Figure 8: Evaluation of the query Q3

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

Figure 9(a) shows the query evaluation time for algorithms NL-NOJ and NL-LOJ. NL-NOJ significantly outperformed NL-LOJ for all queries, except for Q1 and Q2, however the performance of these algorithms on the test queries was not satisfactory. For example, for the quite simple Q3, the NL-NOJ based evaluation took 13,755 s and the NL-LOJ based evaluation took 16,262 s. These algorithms are to be used for joins with high selectivity factors as we show later in this section.

The NL-NOJ and NL-LOJ performance for individual test queries is elaborated in the following. Query Q1 was evaluated in 34 s by both algorithms. The algorithms showed similar performance for Q1 and Q2, since these queries have no nested **OPTIONALS**, and therefore NL-NOJ had no any advantage over NL-LOJ. Queries Q3-Q9 (except perhaps Q7) showed substantial advantage of NL-NOJ over NL-LOJ. The performance difference for Q4 and Q5 is relatively small, because the last input (twin) relation in Q5 is small. NL-NOJ was slightly faster (54 s difference) than NL-LOJ for Q7, since the $Cardinality_o$ of the intermediate twin relation (see Table 3) is not significant. The NL-NOJ based evaluation of Q6 and Q9 was roughly two times faster than the corresponding NL-LOJ based evaluation; the reason is in the large $Cardinality_o$ of the intermediate twin relations and relatively large number of performed joins that additionally require the **NOT NULL** check for the LOJ based evaluation.

4.5 Experiment II: Comparison of SM-NOJ and SM-LOJ

Figure 9(b) shows the query evaluation time for algorithms SM-NOJ and SM-LOJ. SM-NOJ significantly outperformed SM-LOJ for all queries, except for Q1 and Q2. The performance difference between SM-NOJ and SM-LOJ is relatively smaller than the corresponding difference between NL-NOJ and NL-LOJ (e.g., Q6

and Q9), because the sort-merge implementations have better lower bound than the corresponding nested-loops implementations, and the SM-NOJ and 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 0.12 s by both algorithms.

4.6 Experiment III: Comparison of SH-NOJ and SH-LOJ

Figure 9(c) shows the query evaluation time for 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 it 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 0.06 s by both algorithms.

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

Figure 9(d) shows the query evaluation time for 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 foactors: SH-NOJ behaves close to the linear lower bound $\Omega(n)$; SM-NOJ behaves close to the non-linear lower bound $\Omega(n \log n)$; and NL-NOJ has the quadratic lower bound $\Omega(n^2)$.

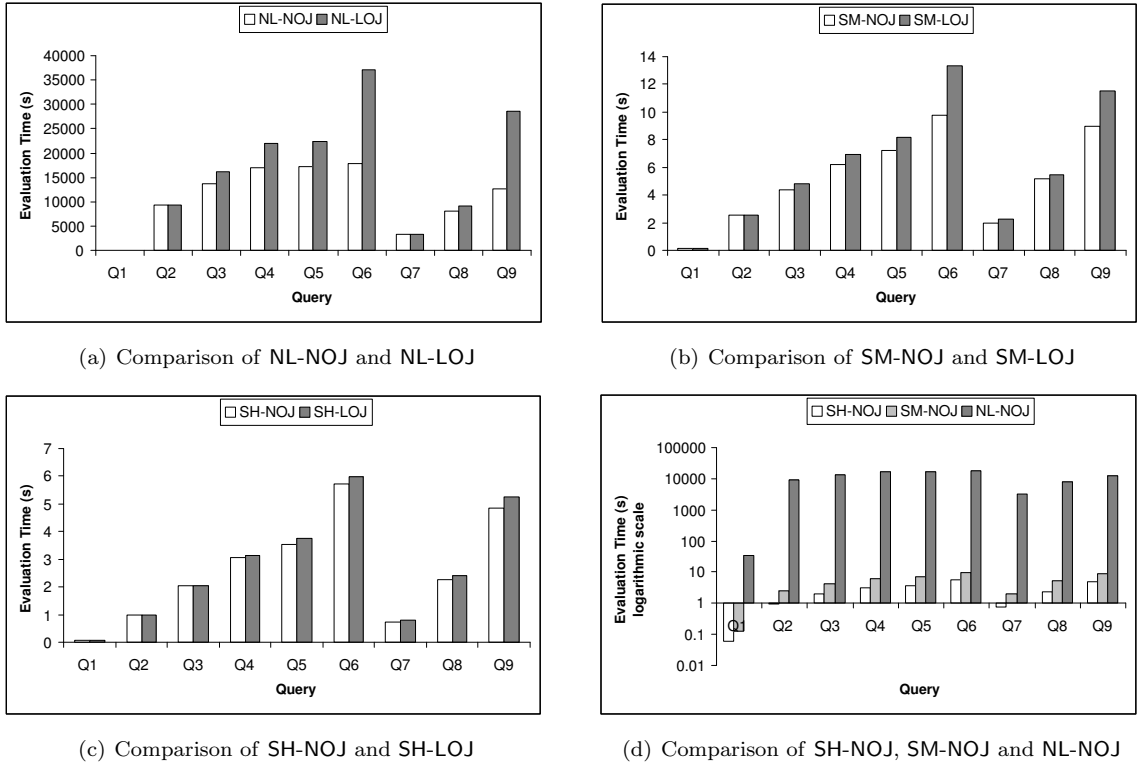


Figure 9: Evaluation of the NOJ and LOJ algorithms using the WordNet test queries

4.8 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

$$JSF = \frac{|(R_1, R_2) \bowtie (S_1, S_2)|}{|(R_1, R_2)| \times |(S_1, S_2)|},$$

where $|(X_1, X_2)|$ means the cardinality of twin relation (X_1, X_2) and $|(X_1, X_2)| = |X_1| + |X_2|$.

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 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 $0.005 \leq JSF \leq 0.1$, 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 $0.0001 \leq JSF < 0.005$: e.g., SH-NOJ took 0.015 s, SM-NOJ took 0.031 s and NL-NOJ took 10.48 s for $JSF = 0.0001$. Both SH-NOJ and SM-NOJ took 0.219 s for $JSF = 0.005$. SM-NOJ was the fastest for $0.01 \leq JSF < 0.8$, and NL-NOJ was the fastest for $0.8 \leq JSF \leq 1.0$. 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 non-linear $n \log n$ performance to quadratic performance with the growth of JSF, however, at the same time, the sorting cost decreases (e.g., for $JSF = 1.0$, input twin relations are already sorted), and (3) NL-NOJ has constant quadratic performance, but does not require neither hashing nor sorting.

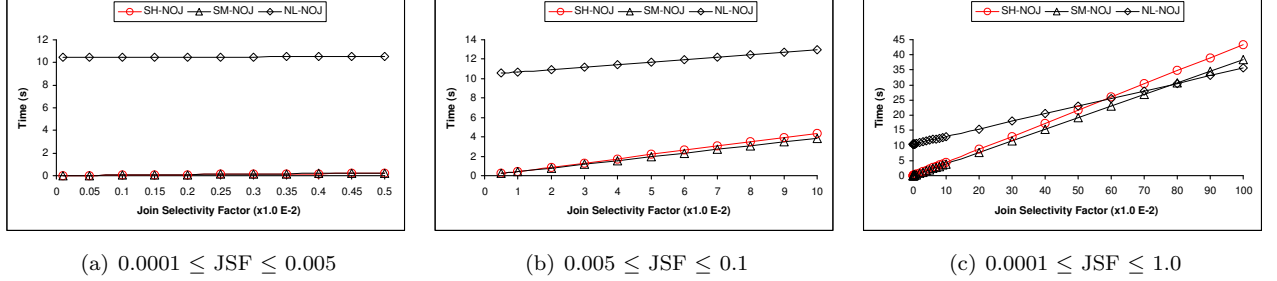


Figure 10: Effect of join selectivity on the NOJ algorithm performance

4.9 Summary

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

- The nested optional join, $(R_b, R_o) \bowtie (S_b, S_o)$, has the performance advantage over the left outer join counterpart when used to evaluate nested `OPTIONALS`, because (1) R_o is always processed in linear time by a NOJ algorithm and (2) the NOJ does not require the `NOT NULL` check. Our experiments on the real-life data set showed that this advantage is significant.
- For NOJs with $JSF \leq 0.005$, SH-NOJ should be used for in-memory evaluation.
- For NOJs with $JSF \geq 0.8$, NL-NOJ should be used for in-memory evaluation.

- For NOJs with $0.005 < JSF < 0.8$, SM-NOJ should be used for in-memory evaluation.

5 Related Work

The join operation, as defined in the relational data model [11, 12], 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 [20, 16] and implemented in RDBMSs. We introduce a new type of join, nested optional join, whose semantics mimics the semantics of optional graph patterns in SPARQL [4]. The NOJ is defined on two twin relations, where each twin relation contains a base relation and an optional relation; therefore, the NOJ can be viewed as a join of four conventional relations. The result of the NOJ is also a twin relation, whose base relation contains tuples that have been concatenated and optional relation – tuples that have been NULL padded. The above semantic and structural characteristics of the NOJ differentiate it from any other join defined in the literature. We propose the NOJ as a favorable alternative to the LOJ-based implementations for the nested optional graph pattern processing with relational databases. Note that the NOJ operator is not a replacement of the LOJ operator: their semantics are different, such as the LOJ needs a special NOT NULL check to return similar results to the NOJ results and this check is not part of the NOJ.

The join processing in relational databases has been an important research for over 30 years and the related literature is abundant [23]. To design algorithms for the NOJ, we use three classical methods for implementing joins in RDBMSs: nested-loops, sort-merge and hash-based join methods [20, 16]. These methods have numerous optimizations including the use of an index which are out of our scope in this paper.

Numerous researchers [13, 9, 26, 25, 18, 28, 8, 27, 24] have proposed to use RDBMSs to store and query RDF metadata 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 OPTIONALS [13, 9]. 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 ... does not fully reproduce SPARQL semantics” [13]. Our work [9] presents algorithm SPARQLtoSQL that provides the first semantics preserving solution to the SPARQL-to-SQL translation problem in the presence of arbitrary complex optional graph patterns. Algorithm SPARQLtoSQL can be extended to use the NOJ operation if it is supported by a RDBMS.

It is worthwhile to mention that SPARQL is not the only RDF query language that supports optional graph patterns. Other examples include SeRQL [5] and RDFQL [2], and the NOJ can be useful for these languages, too.

Finally, a couple of researchers proposed to extend SQL or RDQL [3] to support the query of RDF data. Chong et al. [10] 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. [19] study the problem of RDF aggregate queries by extending the RDQL query language with the `GROUP BY` clause and several aggregate functions, such as *max* and *count*.

6 Conclusions and Future Work

We have proposed a novel relational operator, nested optional join, that enables efficient processing of Semantic Web queries with nested optional patterns. The computational advantage of the NOJ over the currently used LOJ-based implementations comes from the two superior characteristics of the NOJ: (1) the NOJ allows the processing of the tuples that are guaranteed to be NULL padded very efficiently, in linear time and (2) the NOJ does not require the NOT NULL check to return correct results. To facilitate the implementation of the NOJ in relational databases, we have 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. Our empirical study based on the real-life RDF data has verified the efficiency of our algorithms

and has proved the NOJ advantage over the LOJ-based implementations. The experimental results are very promising; the NOJ is a favorable alternative to the LOJ-based evaluation of nested optional patterns in the Semantic Web query processing with RDBMSs. Based on both our theoretical analysis and empirical study, we have provided our recommendations for the use of the presented NOJ algorithms depending on the join selectivity factor: for NOJs with $JSF \leq 0.005$, SH-NOJ should be used for in-memory evaluation; for NOJs with $JSF \geq 0.8$, 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.

In future, we will focus on the following research problems. First, our SPARQLtoSQL algorithm should be extended to support the NOJ operator. Second, index-based implementations of the NOJ should be explored. Finally, the possibility of definition of a “new” relational algebra for RDF query processing should be explored.

References

- [1] RDF Primer. W3C Recommendation, 10 February 2004. <http://www.w3.org/TR/rdf-primer/>.
- [2] RDFQL database command reference. <http://www.intelldimension.com/pages/rdfgateway/reference/db/default.rsp>.
- [3] RDQL - A Query Language for RDF. W3C Member Submission, 9 January 2004. <http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/>.
- [4] SPARQL Query Language for RDF. W3C Working Draft, 4 October 2006. <http://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004/>.
- [5] User guide for Sesame. Updated for Sesame release 1.2.3. <http://www.openrdf.org/doc/sesame/users/index.html>.
- [6] WordNet, a lexical database for the English language. <http://wordnet.princeton.edu/>.
- [7] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.
- [8] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *ISWC*, 2002.
- [9] 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>.
- [10] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, 2005.
- [11] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [12] E. F. Codd. Relational completeness of data base sublanguages. In: *R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California, 1972.*
- [13] R. Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170. 2005. <http://www.hp1.hp.com/techreports/2005/HPL-2005-170.html>.
- [14] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *VLDB*, pages 151–164, 1985.
- [15] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, 1984.
- [16] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2004.

- [17] R. H. Gerber. Dataflow query processing using multiprocessor hash-partitioned algorithms. 1986. Dissertation, University of Wisconsin-Madison, Computer Sciences Technical Report No. 672.
- [18] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. In *SSWS*, 2005.
- [19] E. Hung, Y. Deng, and V. S. Subrahmanian. RDF aggregate queries and views. In *ICDE*, 2005.
- [20] M. Kifer, A. Bernstein, and P. M. Lewis. *Database Systems: An Application Oriented Approach*. Addison-Wesley, 2006.
- [21] W. Kim. A new way to compute the product and join of relations. In *SIGMOD*, pages 179–187, 1980.
- [22] H. Lu, K.-L. Tan, and M.-C. Shan. Hash-based join algorithms for multiprocessor computers with shared memory. In *VLDB*, pages 198–209, 1990.
- [23] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.
- [24] Z. Pan and J. Heflin. DLDB: Extending relational databases to support Semantic Web queries. In *PSSS*, 2003.
- [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] R. Volz, D. Oberle, B. Motik, and S. Staab. KAON SERVER -A Semantic Web Management System. In *WWW, Alternate Tracks - Practice and Experience*, 2003.
- [28] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *SWDB*, 2003.