SQL Query Optimization through Nested Relational Algebra

BIN CAO and ANTONIO BADIA
University of Louisville

Most research work on optimization of nested queries focuses on aggregate subqueries. In this article, we show that existing approaches are not adequate for nonaggregate subqueries, especially for those having multiple subqueries and certain comparison operators. We then propose a new efficient approach, the nested relational approach, based on the nested relational algebra. The nested relational approach treats all subqueries in a uniform manner, being able to deal with nested queries of any type and any level. We report on experimental work that confirms that existing approaches have difficulties dealing with nonaggregate subqueries, and that the nested relational approach offers better performance. We also discuss algebraic optimization rules for further optimizing the nested relational approach and the issue of integrating it into relational database systems.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages–*Query languages*; H.2.4 [**Database Management**]: Systems–*Query processing*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Nested queries, nested relational algebra, nonrelational query processing

ACM Reference Format:

Cao, B. and Badia, A. 2007. SQL query optimization through nested relational algebra. ACM Trans. Datab. Syst. 32, 3, Article 18 (August 2007), 46 pages. DOI = 10.1145/1272743.1272748 http://doi.acm.org/10.1145/1272743.1272748

1. INTRODUCTION

SQL is the standard language for data retrieval and manipulation in relational databases. One of the most powerful features of SQL is nested queries. A *nested query* is a query that has another query embedded within it. An embedded query

This research was sponsored by NSF under the research grant IIS-009128 and NSF CAREER award IIS-0347555.
A preliminary version of part of this article appeared in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ACM, New York, 2005, pp. 191–202.
Authors' address: Computer Engineering and Computer Science Department, University of Louisville, Louisville, KY 40292; email: {bin.cao, abadia}@louisville.edu.

may appear in the FROM clause, in which case it is called a *derived table*, or in the WHERE or HAVING clause, in which case it is called a *subquery*. The query that contains a subquery is called an *outer query*. Theoretically, a query can have an arbitrary number of subqueries nested within it. Subqueries can be either aggregate or nonaggregate. An *aggregate subquery* contains an aggregate function (MIN, MAX, COUNT, SUM, and AVG) in its SELECT clause; it always returns a single value (perhaps null) as the result. A *nonaggregate subquery* in the WHERE clause can be linked to its outer query by one of the following operators: EXISTS, NOT EXISTS, IN, NOT IN, θ ANY (or θ SOME, SOME is a synonym for ANY), and θ ALL, where $\theta \in \{<, \leq, >, \geq, =, \neq\}$; the result is a set of values (perhaps empty). A subquery can be either independent of the outer query or correlated to the outer query. A *correlated subquery* contains a predicate (called a *correlated predicate*) that references the relation in the outer query; thus, the subquery result depends on each tuple in the outer query. A query that contains a correlated subquery is called a *correlated query*.

The traditional evaluation strategy for correlated queries is *nested iteration* [Selinger et al. 1979], that is, for each tuple in the outer query, the subquery is computed once. Since nested iteration might be very expensive, *query unnesting*, that is, rewriting correlated queries into flat forms, has been proposed [Kim 1982; Ganski and Wong 1987; Dayal 1987; Muralikrishna 1989, 1992; Seshadri et al. 1996b]. Unfortunately, most proposed approaches concentrate on aggregate subqueries; nonaggregate subqueries have to be transformed into aggregate subqueries before evaluation (e.g., Ganski and Wong [1987], Galindo-Legaria and Joshi [2001], and Akinde and Bohlen [2003]). There are a few approaches proposed uniquely to nonaggregate subqueries (e.g., Dayal [1987], Baekgaard and Mark [1995], and Badia [2003b]); but these approaches still have some limitations, especially for queries containing multiple subqueries and null values. Most commonly, queries containing nonaggregate subqueries can not be unnested directly; thus, SQL to SQL transformation is required. However, transformation of ALL or NOT IN subqueries is more complicated due to null values. Partly as a result, no unified approach can be applied to all types of subqueries, and each subquery is evaluated by its own strategy. Thus, performance among different subqueries may differ greatly. Finally, most approaches produce quite complex rewritings, and queries containing multiple subqueries are not considered.

In this article, we focus on queries containing nonaggregate subqueries that appear in the WHERE clause, because optimization of aggregate subqueries has been well established and optimization of nonaggregate subqueries still has some limitations. We propose an efficient and uniform approach, the *nested relational approach*, to evaluating queries containing nonaggregate subqueries. Conceptually, the nested relational approach consists of two steps: first, unnesting a query from top-down; second, computing subqueries from bottom-up. The first step is similar to unnesting aggregate subqueries, thus existing techniques (e.g., Dayal [1987]) can be applied. The second step requires computing the subquery result, which is a set of values (maybe empty), as well as comparing an atomic-valued attribute in the outer query to the subquery result. Such operations are not supported by relational algebra. Since the result of a nonaggregate

subquery is a set of values, which can be modeled as a *set-valued* attribute in the nested relational model, we propose to use *nested relational algebra* to process queries containing nonaggregate subqueries.

The nested relational model and nested relational algebra have been studied extensively [Makinouchi 1977; Jaeschke and Schek 1982; Abiteboul and Bidoit 1984; Thomas and Fischer 1986; Schek and Scholl 1986; Ozsoyoglu et al. 1987; Van Gucht 1987; Gyssens and Van Gucht 1988, 1989; Roth et al. 1988; Colby 1989; Paredaens et al. 1989; Vossen 1991; Levene and Loizou 1993, 1994]. As an extension of the traditional relational model [Codd 1970], the nested relational model allows attributes in a nested relation to have nonatomic values. In addition to increasing the power of relational databases, such extensions can support the requirements introduced by applications involving complex objects. However, such extensions also make the definition of nested relational algebra more complicated. For instance, the join operation in nested relations can have *six cases* because the attributes involved in a join operation can be in different subrelations and different nesting levels [Garani and Johnson 2000]. Furthermore, optimization of nested relational operators is not as easy as optimization of relational operators. A typical example is that the basic feature of commutativity of selections and projections for relational operators does not always hold for nested relational operators. Using nested relational algebra in processing queries containing nonaggregate subqueries makes it possible to express the subquery result as a set-valued attribute in nested relations. On the other hand, we have to deal with the complexity introduced by nested relational operators. To take full advantage of nested relational algebra as well as avoid complexity as much as possible, we propose a *simplified version* of nested relational algebra.

In this article, we define our version of nested relational algebra, propose the nested relational approach to efficiently and uniformly evaluating queries containing nonaggregate subqueries, and study algebraic optimization of operators used in our approach. The novelties of the nested relational approach include:

—It not only allows unnesting nonaggregate subqueries directly without any transformation, but also allows all types of subqueries to be evaluated in a uniform manner.

—It works correctly and efficiently for queries containing multiple subqueries and null values.

—It is not drastically changed by the presence or absence of indexes. If indexes exist, different implementations of join operations (for unnesting subqueries) could lead to more efficient plans. However, if no indexes exist, it is still doable and yields reasonable performance (one caveat is that correlated predicates should include at least one equality).

—It has clear semantics and can be further optimized.

—It can be implemented either on top of the relational database management system (RDBMS) using stored procedures or inside RDBMS by modifying existing algorithms.

The rest of this article is organized as follows. In Section 2, we briefly discuss related work in the area of nested query optimization, and then we use an example query to illustrate why existing approaches have difficulties in evaluating nonaggregate subqueries, and give the motivation for using nested relational algebra on query optimization. We also describe previous research on the nested relational model, and give the reason why we need to define our version of nested relational algebra for query optimization. In Section 3, we recursively define our version of nested relational algebra. As a simplified and extended version of traditional nested relational algebra, our algebra includes traditional nested relational operators such as projection and join, as well as a redefined nest operator and a new linking selection operator. In Section 4, we introduce the nested relational approach, which can be applied to queries containing nonaggregate subqueries of any type and any level. We then discuss alternative nested relational approaches based on their different implementations, and analyze the costs of these alternatives. In Section 5, we optimize the nested relational approach by proposing and proving a series of algebraic transformation rules. In Section 6, we further optimize some special queries to gain better performance. In Section 7, we discuss our experiments on different kinds of queries derived from the TPC-H benchmark [Transaction Processing Performance Council] using the nested relational approach. Our experiments cover most issues discussed in Sections 4, 5, and 6, and verify the efficiency and uniformness of the nested relational approach. Finally, we conclude the article in Section 8.

2. RELATED WORK AND MOTIVATION

In this section, we summarize related work in two areas: nested query optimization and the nested relational model, and motivate our work.

2.1 Related Work on Nested Query Optimization

Nested query optimization has received significant attention since the 1980s. Based on the observation that executing correlated queries using the traditional nested iteration method [Selinger et al. 1979] can be very inefficient, Kim [1982] develops query transformation algorithms to rewrite nested queries into equivalent, flat queries which can be processed more efficiently. Ganski and Wong [1987] point out several problems of Kim's algorithms and provide solutions. Dayal [1987] refines and extends Kim's, Ganski and Wang's work to a unified approach to processing queries that contain nested subqueries, aggregates and quantifiers, which enables unnesting queries with more than one nesting level. Muralikrishna [1989] extends Dayal's approach to enable processing the queries that have an arbitrary number of blocks nested within any given block. Seshadri et al. [1996b] propose *magic decorrelation* to nested query optimization based on the *magic sets* rewriting transformation [Mumick et al. 1990], which is implemented in the Starburst database system [Mumick and Pirahesh 1994]. Seshadri et al. [1996a] discuss the theory and practice of applying the magic sets rewriting optimization in a cost-based manner. Although query rewriting may introduce additional operations, we can still expect

to get better performance since a correlated subquery only needs to be executed once.

On the other hand, algebra based approaches have been studied. Ceri and Gottlob [1985] develop a translator that transforms SQL queries into relational algebra extended with aggregate functions. Bultingsloewen [1987] proposes to optimize SQL queries having aggregate functions by transforming SQL queries into relational calculus first, and then into relational algebra. Baekgaard and Mark [1995] propose to incrementally compute nested queries based on the algebra-to-algebra transformation algorithm which makes intensive use of the set-difference operator.

Other approaches involve introducing new operators beyond relational algebra. Galindo-Legaria and Joshi [2001] use the *Apply* operator to evaluate correlated queries in Microsoft SQL Server by decomposing subqueries into primitive, orthogonal pieces. Akinde and Bohlen [2003] propose to transform nested queries into an algebra extended with the *generalized multi-dimensional join* operator (GMDJ) [Chatziantoniou et al. 2001; Akinde and Bohlen 2001], which allows a series of aggregate functions over different conditions using a single algebraic operator.

2.2 Motivation

Although significant research efforts have been devoted to nested query optimization, most proposed approaches focus on aggregate subqueries. Existing strategies for evaluating nonaggregate subqueries are usually based on those for aggregate subqueries. Optimization of nonaggregate subqueries is first studied by Kim [1982]. Kim [1982] transforms `IN` subqueries into *join*[1] operations on the relations in the outer query and the subquery. Ganski and Wong [1987] extend Kim's approach by transforming `EXISTS` or `NOT EXISTS` subqueries to aggregate subqueries with `COUNT`, and `ANY` or `ALL` subqueries to aggregate subqueries with `MIN` or `MAX`, and then using Kim's approach to process the transformed queries. Transforming nonaggregate subqueries to aggregate subqueries also include the work done by Galindo-Legaria and Joshi [2001] and Akinde and Bohlen [2003]. Galindo-Legaria and Joshi [2001] rewrite nonaggregate subqueries as aggregate subqueries with `COUNT`, and evaluate the transformed query by a second-order *Apply* operator. Specifically, `EXISTS` or `NOT EXISTS` subqueries can be further optimized by the *Apply-semijoin* operator or the *Apply-antisemijoin* operator. Akinde and Bohlen [2003] do the similar work but with the GMDJ operator used.

Dayal [1987] proposes to use *semijoin*[2] instead of join in Kim's approach for `IN` subqueries to avoid duplicate output, as well as for `EXISTS` subqueries, and *antijoin*[3] for `NOT EXISTS` subqueries. Dayal [1987] also proposes an alternative

[1]The *join* of relations r and s on condition C is defined as the combined tuples from relations r and s, where tuples in r have matching tuples in s that satisfies C.
[2]The *semijoin* of relations r and s on condition C is defined as the subset of relation r tuples which have matching tuples in s that satisfies C.
[3]The *antijoin* of relations r and s on condition C is defined as the subset of relation r tuples which have no matching tuples in s that satisfies C.

approach to evaluating EXISTS or NOT EXISTS subqueries by performing join operations on the relations in the outer query and the subquery for EXISTS subqueries, or *left outer join*[4] for NOT EXISTS subqueries, and then computing the boolean valued aggregate function EXISTS or NOTEXISTS for each tuple of the outer query, followed by selecting EXISTS=true or NOTEXISTS=true. This strategy also works for ANY, NOT IN, ALL subqueries if they are first transformed into EXISTS or NOT EXISTS subqueries.

However, no matter which transformation is used, special attention must paid to transform NOT IN or ALL subqueries when null values are present. For instance, let $r1$ and $r2$ be relations, A, B be the attributes of $r1$, and C, D be the attributes of $r2$, consider the following query:

```
select  r1.A, r1.B
from    r1
where   r1.B > all (select  r2.D
                    from    r2
                    where   r2.C = r1.D)
```

Assume that for a certain tuple in $r1$ that has several matching tuples in $r2$, that is, $r2.C = r1.D$ is satisfied, the value of $r1.B$ is 5 and the subquery result (denoted by {r2.D}) has a set of values {2, 3, 4, null}. To compute $r1.B > ALL\{r2.D\}$, clearly, 5 is greater than 2, 3, and 4, but this predicate does not return true due to the null value; thus, the current tuple is not the answer. However, if we perform an antijoin of $r1$ and $r2$ on $r1.B <= r2.D$, since no tuples in $r2$ satisfy the condition, the current tuple becomes the answer. This causes conflict. Furthermore, with null present, it is not equal to 0 = (select count($r2.D$)$\cdots$) with the condition $r1.B <= r2.D$ added into the subquery. Akinde and Bohlen [2003] propose to transform an ALL subquery to *two* aggregate subqueries with count(*) and one should be equal to the other. The conditions of the first subquery are the same as the conditions of the original subquery, and the conditions of the second subquery include the conditions of the first subquery plus the condition $r1.B > r2.D$. Although this transformation rule works for queries with null, the transformed query may not be processed efficiently without using special operators like the GMDJ operator. It is getting worse when a query has multiple subqueries. Next, we give an example query to illustrate these problems. We use $r1$, $r2$, and $r3$ as relations. A, B, C, D are the attributes of $r1$, E, F, G, H, I are the attributes of $r2$, and J, K, L are the attributes of $r3$.

Query 1. Motivating example.

```
select  r1.B, r1.C, r1.D
from    r1
where   r1.A > 10 and
        r1.B not in (select  r2.E
```

[4]The *left outer join* of relations r and s on condition C is defined as the union of the join of r and s on C and the unjoined tuples in r padded with null values on the attributes of s.

```
from    r2
where   r2.F = 5 and r2.G = r1.D and
        r2.H > all (select  r3.J
                    from    r3
                    where   r3.K = r1.C and
                            r3.L <> r2.I))
```

A query is a *multi-level* query if it has multiple subqueries. A query is a *one-level* query if all of its subqueries are flat queries which have no subqueries; a query is a *two-level* query if at least one of its subquery is a one-level query, and so on. Thus, Query 1 is a two-level query. From top-down, the second query block is correlated to the first query block by $r2.G = r1.D$, and the third query block is correlated to the other two query blocks by $r3.K = r1.C$ and $r3.L <> r2.I$.

We introduce the term *linking predicate* to refer to the predicate that connects an outer query and its subquery. We call EXISTS, ANY (or SOME), and IN *positive linking operators*, and NOT EXISTS, ALL, and NOT IN *negative linking operators*. If a query has both positive and negative linking operators, we say it has *mixed linking operators*. Thus, Query 1 has two negative linking operators, NOT IN and ALL.

Unnesting Query 1 using existing techniques presents several problems. Note that Query 1 can not be unnested directly; instead, rewriting NOT IN and ALL subqueries is required. However, as explained earlier, simply rewriting such subqueries as aggregate subqueries with COUNT may not preserve semantics when null values are present. Although rewriting is possible (see, for instance, Akinde and Bohlen [2003]), the transformed query is more complicated. In the mentioned approach, two more query blocks have to be added. Even with the efficient GMDJ operator used, the evaluation strategy might not be efficient. In general, the resulting query tree may have several outer joins and antijoins that cannot be moved (except under certain circumstances; see Galindo-Legaria and Rosenthal [1997] and Rao et al. [2001] for approaches to deal with this problem), as well as extra operations. Even though Muralikrishna [1992] proposes to extract (left) antijoins from (left) outer joins, we note that in general such reuse may not be possible: the outer join is introduced to deal with the correlation, and the antijoin with the linking predicate; therefore, they have distinct, independent conditions attached to them. Also, *magic decorrelation* [Seshadri et al. 1996b] would be able to improve the above plan by pushing selections down to the relations; however, this approach does not improve the overall situation, with outer joins and antijoins still present. Also, when (outer) joins are introduced to deal with correlations, all correlated subqueries become one query block. However, when dealing with negative or mixed linking predicates, this creates a problem. To see why, note that in Query 1 we have to outer join $r1$ with $r2$ and $r3$ to determine which tuples of $r3$ must be tested for the ALL linking predicate. However, if the set of tuples in $r3$ related to a tuple in $r1$ and $r2$ fail the test, we can not throw the whole set away. The reason is that if some tuples in $r2$ fail to qualify for an answer, they may make true the NOT IN linking predicate, and hence qualify the $r1$ tuple. Thus, tuples in $r2$ and $r3$ should be antijoined separately to determine which tuples in $r2$ pass or fail the

ALL test. Then the result should be separately antijoined with $r1$ to determine which tuples in $r1$ pass or fail the NOT IN test.

In summary, existing approaches have difficulties in dealing with nonaggregate subqueries, especially for multi-level queries, null values, and negative linking operators. Such operators pose a challenge to most approaches. For instance, in the work of Akinde and Bohlen [2003], a one-level query with a IN subquery requires one aggregate to be computed, but for a NOT IN subquery, it requires two aggregate computations. The situation becomes more complicated for multiple subqueries. Examining all linking predicates involved in nonaggregate subqueries, it is clear that all computations are based on *set* theory. Thus, we argue that all types of subqueries can be evaluated in a uniform manner by simply performing set computations. In this article, our goal is to look for an approach which can uniformly evaluate queries containing nonaggregate subqueries with any type of linking predicates and any level of nesting. In our prior work, *Boolean aggregates* are proposed [Badia 2003b]. When combined with grouping, Boolean aggregates can evaluate nonaggregate subqueries in a uniform manner. In this article, we propose to use *nested relational algebra* because it explicitly represents the intuition that for a given tuple, a nonaggregate subquery provides a set of values (perhaps empty). Thus, linking predicates become set comparison predicates, which can be represented in a straightforward manner in nested relational algebra. Using nested relational algebra to evaluate queries containing nonaggregate subqueries has been studied by Baekgaard and Mark [1995], where nested queries are transformed to flat queries first, and then the flat queries are incrementally computed. The transformation is based on their algebra-to-algebra transformation algorithm which makes intensive use of the relational set-difference operator. Then the set-difference operator is incrementally computed based on view point caches. However, only nested comparison queries, set-membership queries, and set-inclusion queries are discussed. Some researchers have adopted nested relational techniques as new approaches for other database tasks, for instance, generating XML schema from relational databases [Lee et al. 2001; Badia 2003a]. However, optimization is not discussed in such approaches, which concentrate on dealing with the complexities derived from the difference in structure between XML and relations. In this article, we propose an approach based on nested relational algebra to evaluate queries containing nonaggregate subqueries, and discuss possible optimization issues.

2.3 Related Work on Nested Relational Model

The nested relational model is first introduced by Makinouchi [1977] to relax the first normal form (1NF) assumption of the traditional relational data model [Codd 1970]. Jaeschke and Schek [1982] extend the relational model by allowing relations to have set-valued attributes and by introducing two restructuring operators, nest and unnest, to manipulate nested relations. Thomas and Fischer [1986] generalize Jaeschke and Schek's model by allowing nested relations to have relation-valued attributes and arbitrary depth. Since then a number of research efforts have been devoted to the nested relational model [Abiteboul

and Bidoit 1984; Schek and Scholl 1986; Ozsoyoglu et al. 1987; Van Gucht 1987; Gyssens and Van Gucht 1988, 1989; Roth et al. 1988; Colby 1989; Paredaens et al. 1989; Vossen 1991; Levene and Loizou 1993, 1994]. Some of them propose nested relational algebra or calculus to manipulate nested relations. For instance, Abiteboul and Bidoit [1984] present the Verso model, the data structures and operations in Verso. In the Verso model, data is organized in non-1NF relations. The values for some attributes in a Verso relation are atomic whereas the values of other attributes are simpler Verso relations. Furthermore, the atomic attributes in a Verso relation serve as a key. The operations in Verso are recursively defined and include four binary operations (fusion (i.e. union), difference, join, Cartesian product), and five unary operations (projection, selection, restriction, renaming, and restructuring). Colby [1989] defines a recursive algebra for nested relations that allows tuples at all levels of nesting in a nested relation to be accessed and modified directly. The operators of the nested relational algebra can be applied to both relations and subrelations of a relation. Levene and Loizou [1993] introduce the null extended nested relational algebra which incorporates null into the definitions of the null extended operators, and defined a class of integrity constraints that hold in nested relations.

However, for the purpose of SQL query processing, existing versions of nested relational algebra are quite complex. For instance, the join operation in nested relations can have six cases [Garani and Johnson 2000]; for SQL queries, only limited kinds of join operations happen. Most importantly, for SQL query processing, we always start from flat relations and the final result is also a flat relation; nested relations are only used to store intermediate results and perform intermediate operations. However, existing nested relational algebra may have problems when generating a nested relation from a flat relation and later on unnesting the nested relation to the original flat relation. To take advantage of the powerful features of nested relational algebra, as well as to avoid complicated features, we define an algebra dedicated to SQL query processing by simplifying and extending the traditional nested relational algebra. Similar to the Verso model and Colby's algebra, we recursively define our algebra. With respect to SQL semantics, our algebra allows null values and duplicate values. Although our algebra is less powerful than the traditional nested relational algebra, it is easily manageable and efficient for SQL query processing.

3. EXTENDED NESTED RELATIONAL ALGEBRA

In this section, we define our version of nested relational algebra for SQL query processing. Being a simplified version of the traditional nested relational algebra, our algebra include standard operators such as projection and join. For the purpose of SQL query processing, we need to redefine the *nest* operator and introduce the *linking selection* operator. With respect SQL semantics, our algebra allows null values and duplicate values. We assume each relation has a unique non-null attribute served as the primary key (denoted by #). In our definition, we use three-valued logic, that is, the result of a predicate is either `true`, `false`, or `unknown`.

Definition 3.1. Let $U = \{A_1, \ldots, A_n\}$ be a finite set of attributes. A *relation schema* over U is defined recursively as follows:

(1) If $A_1, \ldots, A_n$ are atomic-valued attributes from U, then $R = (A_1, \ldots, A_n)$ is a relation schema over U. The attributes of R are denoted by $attr(R) = \{A_1, \ldots, A_n\}$. The depth of R is 0, denoted by $depth(R) = 0$.
(2) If $A_1, \ldots, A_n$ are atomic-valued attributes from U, $R_1, \ldots, R_m$ are distinct names of relation schemas with a set of attributes (denoted by $attr(R_1), \ldots, attr(R_m)$) such that $\{A_1, \ldots, A_n\}$ and $\{attr(R_1), \ldots, attr(R_m)\}$ are pairwise disjoint, then $R = (A_1, \ldots, A_n, R_1, \ldots, R_m)$ is a relation schema, and $R_1, \ldots, R_m$ are called subrelation schemas or relation-valued attributes. The atomic-valued attributes of R are denoted by $Aattr(R) = \{A_1, \ldots, A_n\}$; the relation-valued attributes of R are denoted by $Rattr(R) = \{R_1, \ldots, R_m\}$; the attributes of R are denoted by $attr(R) = \{A_1, \ldots, A_n\} \bigcup_{i=1}^{m} attr(R_i)$. The depth of R is defined as: $depth(R) = 1 + \max_{i=1}^{m} depth(R_i)$.

Definition 3.2. Let R denote a relation schema over a finite set U of attributes. The *domain* of R, denoted by $DOM(R)$, is defined recursively as follows:

(1) If $R = (A_1, \ldots, A_n)$, where $A_1, \ldots, A_n$ are atomic-valued attributes. Let $DOM'(A_i) = DOM(A_i) \cup \{\texttt{null}\}$, where $1 \leq i \leq n$, and `null`, called the null value, denotes *value unknown*, *value does not exist*, or *no information* [Roth et al. 1989]. Then $DOM(R) = DOM'(A_1) \times \cdots \times DOM'(A_n)$, where "$\times$" denotes the Cartesian product operator.
(2) If $R = (A_1, \ldots, A_n, R_1, \ldots, R_m)$, where $A_1, \ldots, A_n$ are atomic-valued attributes, and $R_1, \ldots, R_m$ are relation-valued attributes. Let $DOM'(A_i) = DOM(A_i) \cup \{\texttt{null}\}$, and $DOM'(R_j) = DOM(R_j) \cup \{\texttt{null}\}$, where $1 \leq i \leq n$ and $1 \leq j \leq m$. Then $DOM(R) = DOM'(A_1) \times \cdots \times DOM'(A_n) \times 2^{DOM'(R_1)} \times \cdots \times 2^{DOM'(R_m)}$, where "$\times$" denotes the Cartesian product operator, and $2^{DOM'(R_j)}$ denotes the power set of the set $DOM'(R_j)$.

A *tuple* t over R is an element of $DOM(R)$. If $A \in attr(R)$, then $t[A]$ denotes the value of t in the column corresponding to A. A *relation* r over R is a finite set of tuples over R. If R contains only atomic-valued attributes, r is called a *flat relation*; otherwise, r is called a *nested relation*.

Definition 3.3. Let r be a relation over the relation schema R. The *projection* of r with respect to L, $\pi_L(r)$, where L is a set of attributes of R, is defined as follows:

(1) If L is empty, then
$\pi(r) := r$
(2) If $L = R_1L_1, \ldots, R_nL_n$, where $R_i (1 \leq i \leq n) \in attr(R)$, $L_i (1 \leq i \leq n)$ is a set of attributes of R_i (L_i is empty if R_i is an atomic attribute), then

$$\pi_{R_1L_1,\ldots,R_nL_n}(r) := \{t \mid \exists t' \in r \wedge t[R_1] = \pi_{L_1}(t'[R_1]) \wedge \vdots \; t[R_n] = \pi_{L_n}(t'[R_n])\}$$

Note that our projection definition keeps duplicate tuples.

The join operation in nested relations is complicated since the attributes involved in a join condition can be in different subrelations and at different nesting levels. Several researchers [Abiteboul and Bidoit 1984; Roth et al. 1988; Colby 1989; Levene and Loizou 1993; Liu and Ramamohanarao 1994; Garani and Johnson 2000] have defined the join operation between two nested relations. Abiteboul and Bidoit [1984] give the definition of joining two nested relations that have the same atomic attributes at the top levels. Roth et al. [1988] introduce the extended natural join operation for cases where the relations that participate in the join have common attributes (atomic and relation-valued) at the top levels. Colby [1989] defines a join operation recursively so that a relation can be joined to another relation or any subrelation of the relation. Levene and Loizou [1993] define the null extended join which supports null values. Liu and Ramamohanarao [1994] introduce the P-Join operator, a combination of Cartesian product, selection and projection. Garani and Johnson [2000] propose a generalization of the natural join operation applicable to all joinable nested relations. Depending on whether an attribute is either atomic-valued or relation-valued and on whether it is at the top level or lower levels of the two relations, the join operation is classified into six cases. The six cases can be further grouped into two categories: one involves the joined two nested relations which have atomic attributes in common, and the other concerns the joined two nested relations that have relation-valued attributes in common.

For SQL query processing, the join operation is quite simple because the base relations are flat relations. In other words, we only need a join on two flat relations. Only for the purpose of optimization (to be discussed in Section 5), we need a join on two nested relations, but the attributes involved in a join operation are always atomic attributes in both relations (similar to the work done by Abiteboul and Bidoit [1984]). Thus, only case 1 [Garani and Johnson 2000] is relevant for our purpose. Unlike Garani and Johnson's definition, our join definition specifies join conditions between two relations.

Definition 3.4. Let r and s be two relations over the relation schemas R and S respectively. Then the *join* of r and s on the conditions C, $r \bowtie_C s$, where $C = A_1\theta_1B_1 \wedge \cdots \wedge A_n\theta_nB_n$, $A_i \in Aattr(R)$, $B_i \in Aattr(S)$, $\theta_i \in \{<, \leq, >, \geq, =, \neq\}(1 \leq i \leq n)$, is defined as follows:

$$r \bowtie_C s := \{t \mid \exists t_r \in r \wedge \exists t_s \in s \wedge t_r[A_1]\theta_1 t_s[B_1] \texttt{ is true} \wedge \cdots \wedge t_r[A_n]\theta_n t_s[B_n] \texttt{ is true} \wedge t[attr(R)] = t_r[attr(R)] \wedge t[attr(S)] = t_s[attr(S)]\}$$

Similarly, we can define a left outer join between two nested relations.

Definition 3.5. Let r and s be two relations over the relation schemas R and S respectively. Then the *left outer join* of r and s on the conditions C,

$r \sqsupset\!\!\bowtie\!\!\lhd_C s$, where $C = A_1\theta_1 B_1 \wedge \cdots \wedge A_n\theta_n B_n$, $A_i \in Aattr(R)$, $B_i \in Aattr(S)$, $\theta_i \in \{<, \leq, >, \geq, =, \neq\}(1 \leq i \leq n)$, is defined as follows:

$$
\begin{aligned}
r \sqsupset\!\!\bowtie\!\!\lhd_C s := \{t \mid \exists t_r \in r \wedge \\
&(\exists t_s \in s \wedge t_r[A_1]\theta_1 t_s[B_1] \texttt{ is true} \wedge \cdots \wedge t_r[A_n]\theta_n t_s[B_n] \texttt{ is true} \wedge \\
&t[attr(R)] = t_r[attr(R)] \wedge t[attr(S)] = t_s[attr(S)]) \vee \\
&(\forall t_s \in s \wedge t_r[A_1]\theta_1 t_s[B_1] \texttt{ is false or unknown} \vee \cdots \vee \\
&t_r[A_n]\theta_n t_s[B_n] \texttt{ is false or unknown} \wedge \\
&t[attr(R)] = t_r[attr(R)] \wedge t[attr(S)] = \{null\})\}
\end{aligned}
$$

Note that $t[attr(S)] = \{null\}$ denotes padding the null value to each attribute in S.

The *nest* operator and the *unnest* operator are two important operators in the nested relational algebra. The nest operator makes a flat relation into a nested relation which allows nonatomic valued attribute, and the unnest operator removes nesting from a nested relation. For our purpose, we only need to redefine the nest operator, which is used to make a subquery result as a set.

Definition 3.6. Let r be a relation over the relation schema R. Let $N_1 \subseteq attr(R)$, $N_2 \subseteq attr(R)$. N_1 and N_2 are disjoint. R_1 is a rename of N_2. The *nest* of r, $\upsilon_{N_1,R_1=N_2}(r)$, is defined as follows:

$$
\begin{aligned}
\upsilon_{N_1,R_1=N_2}(r) := \{t \mid \exists t' \in r \wedge t[N_1] = t'[N_1] \wedge \\
t[R_1] = \{t''[N_2] \mid t'' \in r \wedge t''[N_1] = t[N_1]\}\}
\end{aligned}
$$

N_1 is called the set of *nesting attributes*; N_2 is called the set of *nested attributes*.

Note that in the traditional definition, only N_2 is specified, N_1 is understood as $attr(R) - N_2$. Our definition is just syntactically different from the traditional definition; it does not add any expressive power. Our definition has an implicit projection of $N_1 \cup N_2$; it also highlights the connection between nesting and grouping. Also note that if an attribute A in N_1 has the null value, $t[N_1] = t'[N_1]$ and $t''[N_1] = t[N_1]$ in the above definition should be understood as that the attributes in N_1 except A are the same and A is null. Particularly, the tuples with all attributes in N_1 being null values will be nested into one tuple (this is needed to reproduce SQL semantics).

To help understand the above definitions, we give an example. This example corresponds to Query 1 in Section 2.2.

Example 3.1. Assume $r1$, $r2$, and $r3$ are flat relations with the relation schemas $R1 = (A, B, C, D)$, $R2 = (E, F, G, H, I)$ and $R3 = (J, K, L)$ respectively (shown in Figures 1(a), 1(b), 1(c)), where D, I, L are primary keys (denoted by "#") of each relation. The relation $Temp1$ shown in Figure 1(d) is a flat relation obtained by performing a left outer join of $r1$ and $r2$ on the predicate $G = D$, followed by a left outer join with $r3$ on the predicates $K = C$ and $L <> I$.

The relation *Temp*2 shown in Figure 1(e) is a one-level nested relation obtained by performing $\upsilon_{\{B,C,D,E,H,I\},R_1=\{J,L\}}$ on *Temp*1. Note that there is an implicit projection of the nesting and nested attributes. The reason why we keep

A	B	C	$D(\#)$
12	5	2	1
15	4	3	2
16	null	2	3
11	null	5	4

(a) Relation $r1$

E	F	G	H	$I(\#)$
6	5	1	3	1
5	5	1	1	2
3	5	2	5	3
7	5	4	null	4

(b) Relation $r2$

J	K	$L(\#)$
2	2	4
1	2	3
4	3	1
null	4	2

(c) Relation $r3$

A	B	C	$D(\#)$	E	F	G	H	$I(\#)$	J	K	$L(\#)$
12	5	2	1	6	5	1	3	1	2	2	4
12	5	2	1	6	5	1	3	1	1	2	3
12	5	2	1	5	5	1	1	2	2	2	4
12	5	2	1	5	5	1	1	2	1	2	3
15	4	3	2	3	5	2	5	3	4	3	1
16	null	2	3	null	null	null	null	null	null	null	null
11	null	5	4	7	5	4	null	4	null	null	null

(d) $Temp1 = (r1 \supset\!\bowtie_{G=D} r2) \supset\!\bowtie_{K=C \wedge L<>I} r3$

B	C	$D(\#)$	E	H	$I(\#)$	R_1	
						J	$L(\#)$
5	2	1	6	3	1	2	4
						1	3
5	2	1	5	1	2	2	4
						1	3
4	3	2	3	5	3	4	1
null	2	3	null	null	null	null	null
null	5	4	7	null	4	null	null

(e) $Temp2 = v_{\{B,C,D,E,H,I\},R_1=\{J,L\}}(Temp1)$

Fig. 1. Example of *nest*.

D, I, L, the primary keys of $r1$, $r2$, and $r3$, is that they will be used to determining if a corresponding tuple is empty. As stated above, we assume that each relation has a unique non-null attribute served as a primary key. In our case, a primary key with the null value must be padded by a left outer join operation. If a tuple does not match the join condition, the left outer join operation will pad null values on its attributes including the primary key. Thus, the tuple with the primary key being null can be considered empty. Another reason we keep the primary keys of $r1$, $r2$ and $r3$ is that we have to distinguish between an empty tuple with all attributes being null and a tuple with a certain attribute originally null. As a result, our extended relational algebra can be used on relations containing null values without any problem.

Recall that a *linking predicate* refers to the predicate that connects an outer query and its subquery. For nonaggregate subqueries, a linking predicate compares the atomic value of an attribute in the outer query to the subquery result, which is a set of atomic values (perhaps empty). The formal definition of a linking predicate is given below.

Definition 3.7. Let r be a relation over the relation schema R. Let R_i be one relation-valued attribute in R, R_j be another relation-valued attribute in R ($i \neq j$). A *linking predicate* over r is defined as one of:

—$A\ \theta\ L\ \{B\}$, where $A \in Aattr(R)$ or $A \in attr(R_i)$, $B \in attr(R_j)$, and $\theta \in \{<, \leq, >, \geq, =, \neq\}$, $L \in \{\texttt{ANY}, \texttt{ALL}\}$.
—$L\ \{B\}$, where $L \in \{\texttt{EXISTS}, \texttt{NOT EXISTS}\}$ and B as above.

A is called a *linking attribute*, B is called a *linked attribute*, and L is called a *linking operator*. For convenience, `EXISTS` and `ANY` (`IN` is equivalent to "`=ANY`") are called *positive linking operators*, `NOT EXISTS` and `ALL` (`NOT IN` is equivalent to "$\neq$ `ALL`") are called *negative linking operators*. If a query has both positive and negative linking operators, then we say it has *mixed linking operators*.

To evaluate linking predicates, we extend the definition of the standard selection by allowing a linking predicate to be a selection condition.

Definition 3.8. Let r be a relation over the relation schema R. The *selection* of r with respect to C, $\sigma_C(r)$, where C is either a standard predicate with the form of $A\ \theta\ B$, or a linking predicate with the form of $A\ \theta\ L\ \{B\}$ or $L\ \{B\}$ (see Definition 3.7), is defined recursively as follows:

(1) If $A \in Aattr(R)$, then
  —If C is a standard predicate with the form of $A\ \theta\ B$, where $\theta \in \{<, \leq, >, \geq, =, \neq\}$, then
    —If $B \in Aattr(R)$, then
      $\sigma_C(r) := \{t \mid t \in r : t[A]\ \theta\ t[B]\ \texttt{is true}\}$
    —If B is a constant, then
      $\sigma_C(r) := \{t \mid t \in r : t[A]\ \theta\ B\ \texttt{is true}\}$
  —If C is a linking predicate, then
    —If C is of the form $A\ \theta\ L\ \{B\}$, where $B \in attr(R_j)$, $R_j \in Rattr(R)$, $\theta \in \{<, \leq, >, \geq, =, \neq\}$, and L is `ANY`, then
      $\sigma_C(r) := \{t \mid t \in r \wedge \exists t' \in t[R_j] : t[A]\ \theta\ t'[B]\ \texttt{is true}\}$
      If L is `ALL`, then
      $\sigma_C(r) := \{t \mid t \in r \wedge \forall t' \in t[R_j] : t[A]\ \theta\ t'[B]\ \texttt{is true}\}$
    —If C is of the form $L\ \{B\}$, where $B \in attr(R_j)$, $R_j \in Rattr(R)$, B is the primary key of R_j, and L is `EXISTS`, then
      $\sigma_C(r) := \{t \mid t \in r \wedge \exists t' \in t[R_j] : t'[B]\ \texttt{is not null}\}$
      If L is `NOT EXISTS`, then
      $\sigma_C(r) := \{t \mid t \in r \wedge \forall t' \in t[R_j] : t'[B]\ \texttt{is null}\}$

(2) If $A \in attr(R_i)$, where $R_i \in Rattr(R)$, then
$$\sigma_C(r) := \{t \mid \exists t_r \in r \wedge t[attr(R) - \{R_i\}] = t_r[attr(R) - \{R_i\}] \wedge t[R_i] = (\sigma_C(t_r[R_i]) \neq \emptyset)\}$$

If C is a linking predicate, such a selection is called a *linking selection*.

For multi-level queries, a tuple that does not satisfy a selection condition can not be discarded arbitrarily because it may contribute to later processing [Muralikrishna 1989]. We have shown an example in Section 2.2 (see page 7). The strategy to keep this tuple is to pad null values on corresponding attributes

B	C	$D(\#)$	E	H	$I(\#)$
5	2	1	6	3	1
5	2	1	null	null	null
4	3	2	3	5	3
null	2	3	null	null	null
null	5	4	7	null	4

(a) $Temp3 = \pi_{B,C,D,E,H,I}(\sigma'_{H>ALL\{R_1(J)\}\vee R_1(L)\ is\ null,\{E,H,I\}}(Temp2))$

B	C	$D(\#)$	E	H	$I(\#)$
5	2	1	6	3	1
4	3	2	3	5	3
null	2	3	null	null	null
null	5	4	7	null	4

(b) $Temp4 = \pi_{B,C,D,E,H,I}(\sigma_{H>ALL\{R_1(J)\}\vee R_1(L)\ is\ null}(Temp2))$

Fig. 2. Example of *linking selection*.

in this tuple. For this purpose, we introduce *pseudo-linking selection*, which keeps the tuples that pass the condition, as well as the tuples that fail the condition by padding null values on corresponding attributes.

Definition 3.9. Let r be a relation over the relation schema R. The pseudo-linking selection of r with respect to C, $\sigma'_{C,D}(r)$, where $D \subset attr(R)$, C is a linking predicate with the form of $A\ \theta\ L\ \{B\}$ or $L\ \{B\}$ (see Definition 3.7), is defined recursively as follows:

(1) If $A \in Aattr(R)$, then

$$\sigma'_{C,D}(r) := \{t \mid \exists t' \in r \wedge (C(t') \texttt{ is true} \wedge t = t') \vee (C(t') \texttt{ is false or unknown} \wedge t[attr(R) - D] = t'[attr(R) - D]) \wedge t[D] = \{null\})\}$$

(2) If $A \in attr(R_i)$, where $R_i \in Rattr(R)$, then

$$\sigma'_{C,D}(r) := \{t \mid \exists t_r \in r \wedge t[attr(R) - \{R_i\}] = t_r[attr(R) - \{R_i\}] \wedge t[R_i] = \sigma'_{C,D}(t_r[R_i])\}$$

Note that $t[D] = \{null\}$ denotes padding the null value to each attribute in D.

To help understand the definition of linking selection, we give an example that follows Example 3.1.

Example 3.2. The relation *Temp*3 shown in Figure 2(a) is a projection of $B, C, D,\ E, H, I$ on the result of a pseudo-linking selection $\sigma'_{H>ALL\{R_1(J)\}\vee R_1(L)\ is\ null,\{E,H,I\}}$ on *Temp*2 shown in Figure 1. Note that *Temp*3 is a flat relation. A negative linking predicate returns true if the subquery result is empty, which is identified by the primary key being null. Thus, we have additional condition $R_1(L)$ *is null* doing linking selection. Under our definition, even though the linking selection over the second tuple returns false, we can not discard this tuple. We have to keep this tuple by padding null values on E, H, I. The linking selection over all other tuples returns true, thus we

keep these tuples in their original forms. One notable point is that for the fourth and the fifth tuples, although the linking selection compares H(null) to $\{R_1(J)\}(\{\texttt{null}\})$, the linking selection returns true because the result of the condition $R_1(L)$ *is null* is true.

The relation *Temp*4 shown in Figure 2(b) is obtained by a projection of B, C, D, E, H, I on the result of a linking selection $\sigma_{H>ALL\{R_1(J)\}\vee R_1(L)\ is\ null}$ on *Temp*2. The linking selection over the second tuple returns false, thus we discard this tuple. All other tuples pass the linking selection and become the result.

4. NESTED RELATIONAL APPROACH TO SQL QUERY PROCESSING

The motivation of the nested relational approach is based on the observation that a linking predicate is actually a *set* computation. The basic idea of the nested relational approach is straightforward: a nested query is unnested from top-down first, and then the linking predicates are computed from bottom-up.

4.1 Nested Relational Approach

For a nested query with n query blocks, in each query block, from top-down, let r_i $(1 \leq i \leq n)$ denote the relations in the FROM clause; L_i $(1 \leq i \leq n-1)$ denote the linking predicate between blocks i and $i+1$; C_{ij} $(2 \leq i \leq n$ and $1 \leq j \leq n)$ represent the correlated predicate(s) between block i and j $(i > j)$, and Δ_i $(1 \leq i \leq n)$ represent the predicates in the WHERE clause except L_i and C_{ij}. The nested relational approach proceeds in three steps.

(1) First, we reduce each query block to a single relation, called *reduced relation*, by doing all operations in the WHERE clause except the linking predicate and correlated predicate(s), that is, at each block i, produce $T_i = \sigma_{\Delta_i}(r_i)$.[5] Note that this is equivalent to producing the complementary set in the *magic decorrelation* technique [Seshadri et al. 1996b]; however, we do not produce a magic set.

(2) Second, we create a *tree expression* for the query as follows: walk through the query in Depth-First, Left-to-Right order; create one node for each query block. We label each node with the corresponding T_i. Between any two adjacent nodes T_i and T_{i+1}, we add an edge directed from T_i to T_{i+1} labeled with the linking predicate L_i. If T_{i+1} is correlated to T_i, we add the correlated predicate $C_{(i+1)i}$ to the edge. If T_i is correlated to a non-adjacent node T_j $(i > j)$, we add the correlated predicate C_{ij} to the edge between T_i and T_{i-1} if all edges between T_j and T_i have been labeled with correlated predicates; otherwise, we add an edge directed from T_j to T_i labeled with the correlated predicate C_{ij}. The *root* is labeled by the name of the outermost query block, *leaves* are labeled by the name of innermost query blocks, other nodes are labeled by the name of the middle query blocks. A node is called a *subroot* if it has more than one children. All nodes under a subroot are called a *subtree* of the subroot. For a given node n, let $name(n)$ be the T_i that serves as name

[5]We assume that all relations are connected, that is, no Cartesian product present. We also assume that all predicates are linked by AND.

of the node; $link_C(n, m)$ be the C_{ij} (if one exists) and $link_L(n, m)$ be the L_i, which label the link between n and one of his children m.

(3) Third, we compute(*root*, T_1). The algorithm, shown as Algorithm 1, recursively goes down the tree in *depth-first* manner, creating a single relation through the use of join or outer join. Note that the structure created in the previous step may be a graph. In this step, we restrict our attention to edges labeled with correlated predicates, in which case we get a maximal spanning query tree for the graph (when all query blocks are correlated). Noncorrelated subqueries are executed once in the first step, and the result is used by every tuple (*virtual* Cartesian product). When a leaf is reached, the algorithm goes bottom-up nesting the relation obtained and applying a corresponding linking selection to reduce the relation. When a subroot is found on the way down, the algorithm chooses a child to continue towards the leaves; on the way up, however, the algorithm will go down again until all paths in the subtree of the subroot have been covered before proceeding up past the subroot .

Algorithm 1. Compute(node, relational-expression)

input : a nested query with nonaggregate subqueries.
output: the result of a query.

```
1  PROCEDURE compute(node, rel) {
2  if node is a leaf then
3  |  return;
4  else
5  |  foreach n ∈ children(node) do
6  |  |  T_i = name(n);
7  |  |  C_ij = link_C(node, n)
8  |  |  L_i = link_L(node, n)
9  |  |  if C_ij ≠ ∅ then
10 |  |  |  rel = rel ⟕_{C_ij} T_i or rel = rel ⟖_{C_ij} T_i
11 |  |  else
12 |  |  |  rel = rel × T_i
13 |  |  end
14 |  |  compute(n, rel)
15 |  |  rel = υ_{{T_1.*,...},{T_i.*}}(rel)
16 |  |  rel = σ_{L_i}(rel) or σ'_{L_i}(rel)
17 |  end
18 end
19 }
```

The algorithm works equally for nested linear queries and nested tree queries.[6] For linear queries, there is only one child for each node; the net effect is

[6]A nested *linear* query is a query in which at most one query block is nested within any query block. A nested *tree* query is a query in which there is at least one query block that has two or more query blocks nested within it at the same level.

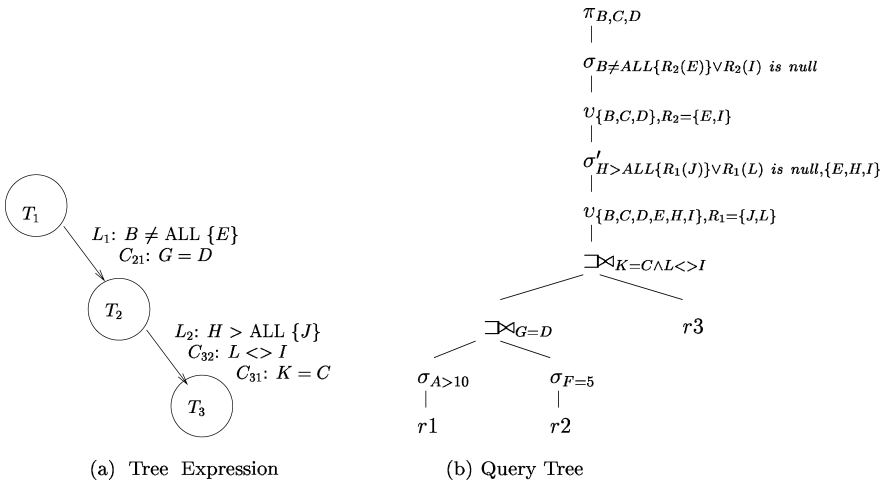


Fig. 3. Nested Relational Approach Applied to Query 1.

that of going down the tree joining or outer joining, or using the Cartesian product when there is no correlation (this Cartesian product is really virtual), and then up nesting and evaluating the predicates. For tree queries, each subroot makes us go down all paths before continuing on the way up.

The following example shows how the nested relational approach processes a nested query, We still use Query 1 in Section 2.2 as our example query.

Example 3.1. The first step is to reduce each query block to reduced relations T_1, T_2 and T_3 by doing all operations in the `WHERE` clause except linking predicates and correlated predicates. Next, we can obtain the tree expression for this query (shown in Figure 3(a)). Based on the tree expression, we process the query from top-down. We use a *query tree* to represent the process of a query evaluation, in which π denotes projection; $\supset\!\bowtie$ left outer join; σ or σ' (linking) selection; υ nest. The query tree for processing Query 1 is shown in Figure 3(b) (intermediate projections are omitted). We start from root node T_1, performing a left outer join of T_1 and T_2 on the correlated predicate $G = D$. Since T_2 is not a leaf, we keep performing a left outer join with T_3 on the correlated predicates $K = C$ and $L <> I$. Since T_3 is a leaf node, we compute the linking predicate L_2: $H > ALL\{J\}$, which is achieved by $\upsilon_{\{B,C,D,E,H,I\},R_1=\{J,L\}}$ followed by a pseudo-linking selection $\sigma'_{H>ALL\{R_1(J)\} \vee R_1(L) \text{ is null},\{E,H,I\}}$. Then, it goes back to node T_2. Since there is no other children under node T_2, we compute the linking predicate $B \neq ALL\{E\}$ (`NOT IN` is equivalent to "$\neq$`ALL`") by $\upsilon_{\{B,C,D\},R_2=\{E,I\}}$ followed by a linking selection $\sigma_{B \neq ALL\{R_2(E)\} \vee R_2(I) \text{ is null}}$, which goes back to root T_1. The final result is obtained by a projection of the desired attributes. In this example, we use pseudo-linking selection for the `ALL` linking predicate, and linking selection for `NOT IN`. Generally, if a query has only positive linking operators, or if the linking predicate is the last one to be computed, we use linking selection. In all other cases, we have to use pseudo-linking selection. From this example, we can see that the nested relational approach treats all linking operators equally, and the resulting query tree is simple.

So far we have assumed that all predicates in the WHERE clause are linked by AND. We call the subquery that is linked to other predicates by AND a *AND-ed subquery*. If a subquery is linked to the correlated predicates (if exist) by AND and linked to other predicates by OR, we call it a *OR-ed subquery*. For OR-ed subqueries, The relations that are involved in the linking predicate and the correlated predicates are the unreduced relations, while the relations that are involved in all other predicates except these two are reduced relations. The query result (including the subquery result) either depends on the linking predicate and the correlated predicates (if exist) or other predicates; if either of them returns true, the query will return something as result. Thus, the nested relational approach needs to be slightly modified to handle such subqueries. First, we still reduce each query block to the reduced relation, as well as keep a copy of the unreduced relations. Second, unlike we always unnest subqueries for AND-ed subqueries, we always consider the reduced relations first for OR-ed subqueries. If the reduced relation is not empty, we do not need to compute the linking predicate and only need to output the desired attributes in the reduced relation as result. It is similar to noncorrelated subqueries. If the reduced relation is empty, we follow the same way as evaluating AND-ed subqueries. Instead of performing joins or outer joins of the reduced relations, we perform joins or outer joins of the unreduced relations on the correlated predicates. and then compute the linking predicate. The strategy for OR-ed subqueries seems not to be efficient because we need to keep copies of unreduced relations. However, dealing with disjunction is always problematic, due to the "liberal" semantics of the OR connector. One advantage of OR-ed subqueries is that we can always compute the reduced relation first, and if it is not empty, we do not need to compute the subquery.

4.2 Implementation Issues

Since the nested relational approach involves new operators, in this section, we discuss possible strategies for implementing these operators. We based our discussion on that the nested relational approach is implemented on top of RDBMS, as we do in our experiments (to be discussed in Section 7). That is, the system does subquery unnesting, and we obtain the intermediate result from the system and implement the required operators. In our implementation, subrelations are assumed to be stored inline with other attributes. If the size of a subrelation is too large, it can be stored in a separate table. We give a rough discussion on this topic at the end of this article.

Examining the query trees shown in Figure 3(b), we can simply divide the nested relational approach (referred to NRA in what follows) into two parts: the *flat* part, which involves selection, join or outer join operators for unnesting a query, and the *nested* part, which involves nest and linking selection operators for computing linking predicates. Since the flat part can be implemented and optimized using existing techniques, only the nested part needs implementation efforts. Similar to the group-by operator, the nest operator can be implemented either by sorting or hashing. Here we assume that sorting is used. The linking selection operator can be implemented in a similar manner to IN (either hashing

or table lookup). Here we assume that scanning is used. Depending on different implementation strategies of the nest and linking selection operators, we have three alternative approaches: Naive NRA, Nest-Merged NRA, and Pipelined NRA.

4.2.1 *Naive NRA.* The nested relational approach requires a nest operator and a linking selection operator to compute each linking predicate. Intuitively, the nest operator can be implemented first, then followed by the linking selection operator. Following this strategy, we have *Naive NRA*. Assume that a query has n query blocks, query block $i (2 \leq i \leq n)$ is nested within query block $i-1$, T_i is the reduced relation of query block i, and $L_i (1 \leq i \leq n-1)$ is the linking predicate that connects query block i and query block $i+1$. We use a relation *temp* to denote the intermediate result of joins or left outer joins of $T_1, \ldots, T_n$, which is sorted on the primary keys of $T_1, \ldots, T_n$. Naive NRA works as follows. To compute the linking predicate L_{n-1}, we nest the relation *temp* into a nested relation $temp_1$ by scanning each tuple in the intermediate result. Since it is sorted on the primary keys of $T_1, \ldots, T_n$, the tuples with the same primary keys on $T_1, \ldots, T_{n-1}$ are grouped together as the nesting attributes, and T_n is nested as the nested attributes (subrelation). Then, for each tuple t_{temp_1} in $temp_1$, we compute L_{n-1} with pseudo-linking selection by comparing the value of the linking attribute (i.e., an attribute in T_{n-1}) to each value of the linked attribute (i.e., an attribute in T_n): if L_{n-1} is not satisfied, we pad null values on the attributes of T_{n-1}. Next, to compute the linking predicate L_{n-2}, we nest $temp_1$ into $temp_2$ by making the attributes of $T_1, \ldots, T_{n-2}$ the nesting attributes, and the attributes of T_{n-1} the nested attributes. Then, for each tuple t_{temp_2} in $temp_2$, we compute L_{n-2} with pseudo-linking selection: if L_{n-2} is not satisfied, we pad null values on the attributes of T_{n-2}. We follow the same way to compute $L_{n-3}, \ldots$, until L_2 (with the nested relation $temp_{n-2}$). Finally, to compute the linking predicate L_1, we nest $temp_{n-2}$ into $temp_{n-1}$ by making the attributes of T_1 the nesting attributes, and the attributes of T_2 the nested attributes. Then, for each tuple $t_{temp_{n-1}}$ in $temp_{n-1}$, we compute L_1 with linking selection: if L_1 is satisfied, we output the desired attributes of $t_{temp_{n-1}}[T_1]$ as a part of the final result; otherwise, we discard $t_{temp_{n-1}}$. Note that computing linking predicates can be optimized; for a specific linking operator, once a comparison makes the linking predicate invalid, we know the linking predicate is not satisfied and we can go ahead to compute the next tuple. For instance, an "= `ALL`" can be skipped once we find a single value that is not equivalent to the linking attribute.

4.2.2 *Nest-Merged NRA.* In Naive NRA, we compute each linking predicate by using one nest followed by one linking selection. For multi-level nested queries, that means for a linking predicate L_i, we have to nest the intermediate result first and then do the linking selection; for the next linking predicate L_{i-1}, we have to nest again and then do the linking selection; and so on. This strategy might be very expensive because each nest operation needs to access the result of the previous step once. However, examining the parameters of two adjacent nest operators in a query tree, for instance, see Figure 3(b), we

notice that these two operators have some relationship. To compute the linking predicate $H > ALL\{J\}$, the nesting attributes are $\{B, C, D, E, H, I\}$; next to compute $B \neq ALL\{E\}$, the nesting attributes are $\{B, C, D\}$, and the nested attributes are $\{E, I\}$, both of which are from the previous nesting attributes. This advantageous feature gives rise to an optimization of Naive NRA: doing all nest operations in a single step, followed by executing the linking selection operations one by one, instead of intertwining nest and linking selection. We call it *Nest-Merged NRA*. It gives a feasible and efficient implementation due to the fact that only the deepest or the first nest operator involves true (physical) reordering of the tuples in the intermediate result, all others are conceptual. We use the same assumptions used in Naive NRA. Nest-Merged NRA works in two steps. First, we nest *temp* into a $n-1$-level nested relation *temp*′ by making the attributes of T_1 the nesting attributes, the attributes of T_2 both the nested attributes of T_1 and the nesting attributes of T_3, until the attributes of T_n the nested attributes of T_{n-1}. Note that all these nest operations can be done in a single step by sorting *temp* on the primary keys of $T_1, \ldots, T_{n-1}$. Second, for each tuple t in *temp*′, we compute L_{n-1} with pseudo-linking selection: if L_{n-1} is not satisfied, we modify the attributes of T_{n-1} with null values. Then we continue to compute L_{n-2}, L_{n-3}, ..., until L_2. Finally, we compute the linking predicate L_1 with linking selection: if L_1 is satisfied, we output the desired attributes of $t[T_1]$ as a part of the final result; otherwise, we discard t.

4.2.3 *Pipelined NRA.* Pipelining is possible in the context of the nested relational approach. In the flat part, selection operators can be pipelined to join or outer join operators. In the nested part, linking selection operators can be pipelined to nest operators provided that the intermediate result is properly ordered. Pipelining the nested part allows two operators to be computed simultaneously. Thus, the cost of such plans can be further reduced even if no modification to the plan takes place. We call the nested relational approach with the linking selection and nest operators pipelined *Pipelined NRA*. We use the same assumptions used in Naive NRA. Similar to Nest-Merged NRA, one important requirement for Pipelined NRA is that the intermediate result *temp* must be sorted on the primary keys of $T_1, \ldots, T_{n-1}$. We classify *temp* into *groups* and *subgroups*. Tuples in *temp* with the same primary key of T_1 belong to a group. In each group, tuples with the same primary key of T_2 belong to a subgroup. In each subgroup, tuples with the same primary key of T_3 belong to another subgroup; and so on. In Pipelined NRA, linking predicates $L_1, \ldots, L_{n-1}$ are computed on the fly with a sequential pass over the intermediate result *temp*. We start from a tuple t_1 in *temp*, cache the values of $t_1[T_1], \ldots, t_1[T_{n-1}]$, and compute L_{n-1} by comparing values involved in $t_1[T_{n-1}]$ and $t_1[T_n]$; if L_{n-1} is satisfied, values of $t_1[T_{n-1}]$ contribute to the candidate value to compute L_{n-2}. Then, we move to the next tuple t_2 in *temp*. If t_2 belongs to the same group and the same subgroups as t_1, we keep computing L_{n-1}, and move to the next tuple. If t_2 belongs to the same group as t_1, but starts new subgroups, we compute linking predicates that correspond to these subgroups. For instance, if all the primary keys of $t_2[T_1], \ldots, t_2[T_{n-2}]$ are the same as those of $t_1[T_1], \ldots, t_1[T_{n-2}]$, but the primary key of $t_2[T_{n-1}]$ is different from the primary key of $t_1[T_{n-1}]$,

we compute L_{n-2} by comparing cached values of $t_1[T_{n-2}]$ and $t_1[T_{n-1}]$, cache the values of the new subgroup, and compute L_{n-1}. If t_2 starts a new group different from t_1, we compute L_1 by comparing cached values of $t_1[T_1]$ and $t_1[T_2]$ ($t_1[[T_2]$ may be replaced by the values of the latest subgroup). If L_1 is satisfied, we output the desired attributes of $t_1[T_1]$ as a part of the final result; otherwise, we discard t_1. We then repeat the above steps to pass the intermediate result *temp* until the last tuple.

We discuss implementation issues based on linear queries. For tree queries that have more than one subqueries at the same level, we can avoid cross products between sibling subqueries. We arbitrarily choose a subquery as a start point; this subquery together with its outer query is a linear query. Then, after we finish computing this subquery and have the result, we use this result to compute the next subquery, which is still a linear query. We follow the same way until all subqueries are computed. When subqueries are AND-ed together, we can further use as starting point the result of the previous computation (not so when they are OR-ed). However, in any case, one can avoid cross products.

4.3 Cost Analysis

In this section, we analyze the cost of the nested relational approach. We consider only I/O costs and measure I/O cost in terms of the number of page I/Os. Theoretically, the cost of the nested relational approach is equivalent to the cost of implementing the flat part plus the cost of implementing the nested part.

For the flat part, we assume that selections are pipelined to (outer) joins; thus, the cost of the flat part is equivalent to the cost of (outer) joins. Assume that a query has n query blocks, query block $i (2 \leq i \leq n)$ is nested within query block $i-1$, r_i is the relation in each query block $i (1 \leq i \leq n)$, M_{r_i} is the size of r_i, and $M'_j (1 \leq i \leq n-2)$ is the size of the result of (outer) joins on two relations. We also assume that hash joins are used. Then the cost of the flat part, denoted by $Cost_{flat}$ is:

$$Cost_{flat} = 3\left(\sum_{i=1}^{n} M_{r_i} + \sum_{j=1}^{n-2} M'_j\right).$$

The cost of the nested part depends on which nested relational approach is used. Based on our assumption that the nest operator is implemented by sorting, one nest costs $2M \log_{B-1} M$ if a $(B-1)$-way merge sort is used [Kim 1982], where B is the available number of buffers in memory.

For Naive NRA, we assume that the size of the result of (outer) joins is M_1, and the size of the relation after each linking predicate is computed is $M_k (2 \leq k \leq n-1)$. Note that $M_1 \geq M_2 \geq \cdots \geq M_{n-1}$. The cost of the nested part using Naive NRA, denoted by $Cost_{nested\_Naive}$, is:

$$Cost_{nested\_Naive} = \sum_{k=1}^{n-1}(2M_k \log_{B-1} M_k + M_k) \leq (n-1)(2M_1 \log_{B-1} M_1 + M_1).$$

Unlike Naive NRA, Nest-Merged NRA (for multi-level queries) needs sorting only once, plus $n-1$ pass over M_1 to compute linking predicates. The cost of

the nested part using Nest-Merged NRA, denoted by $Cost_{nested\_Nest\text{-}Merged}$, is:

$$Cost_{nested\_Nest\text{-}Merged} = 2M_1 \log_{B-1} M_1 + (n-1)M_1.$$

Pipelined NRA needs a sequential pass over M_1 to compute linking predicates. The cost of the nested part using Pipelined NRA, denoted by $Cost_{nested\_Pipelined}$, is:

$$Cost_{nested\_Pipelined} = 2M_1 \log_{B-1} M_1.$$

Note that the cost of the flat part is fixed to Naive NRA, Nest-Merged NRA, and Pipelined NRA. Clearly, the performance comparison among these three alternatives is: Pipelined NRA < Nest-Merged NRA < Naive NRA.

5. ALGEBRAIC OPTIMIZATION

The most important characteristic of the nested relational approach is that it allows the study of algebraic optimization in a manner similar to relational algebra. However, optimization of nested relational operators is more complicated compared to relational operators. Algebraic optimization of nested relational operators have been studied [Scholl 1986; Jan 1990; Liu and Ramamohanarao 1994; Liu and Yu 2005]. Scholl [1986] presents a series of equivalence rules among algebraic expressions containing unnest and reduction operators as well as selections and projections. Jan [1990] discusses the commutative properties of nested relational operators in the context of optimizations. Liu and Ramamohanarao [1994], Liu and Yu [2005] investigate some algebraic properties of their extended nested relational operators and proposed several algebraic transformation rules. They also outline an algorithm that uses those rules to transform an initial query tree into an optimized tree. However, proposed rules are not sufficient for the nested relational approach. As we have known, the nested relational approach only involves selection, (outer) join, nest, (pseudo) linking selection and projection, which can be considered as a simplified and extended version of the standard nested relational algebra. Thus, algebraic optimization of the nested relational approach only needs to be concerned with these operators. We take advantage of the simplicity of our version of nested relational algebra, and of its specific purpose by proposing rules that fit the patterns that appear in the nested relational approach to SQL optimization. Proofs of the proposed rules can be found in the electronic appendix.

5.1 Projection

In the nested relational approach, a projection can always be pushed down to the base relation if the projection does not discard attributes required by later processing.

LEMMA 1 (PUSHING PROJECTION DOWN TO BASE RELATION). *Let r and s be two relations over the flat schemas R and S respectively. Let $A\theta B$ be a join condition between r and s, where $A \in attr(R)$, $B \in attr(S)$ and $\theta \in \{<, \leq, >, \geq, =, \neq\}$. If $X \subseteq attr(R)$, $Y \subseteq attr(S)$, then $\upsilon_{\{X\},R_1=\{Y\}}(r \rtimes_{A\theta B} s) = \upsilon_{\{X\},R_1=\{Y\}}(\pi_{X,A}(r) \rtimes_{A\theta B} \pi_{Y,B}(s))$.*

Intuitively, Lemma 1 states that the attributes required by the nest operator (X and Y) can be projected on base relations. Pushing down projections is efficient because it permits reductions in the size of X and Y by eliminating unnecessary attributes as early as possible. In this article, we always push down projections whenever possible.

5.2 Nest and Join

The nested relational approach decorrelates subqueries using (outer) joins of relations in each query block. These join operations might be very expensive and the result of the join operations might be very large for later processing. If we can reduce the number of tuples in a relation that participates in a join operation, we might reduce the cost of join. To achieve this goal, one possible solution is to push the nest operation down before (outer) join. However, this is not always possible; the conditions under which it can be done are similar to the conditions to push down a group-by operator past a join [Gupta et al. 1995; Yan and Larson 1994].

Pushing down nest makes the flat relation in a subquery to be a nested relation. Accordingly, the join operation become a join of a flat relation and a nested relation, but the join attributes still atomic attributes in both relations. To push down the nest operation, the nesting attributes should be the correlated attributes (the attributes of the subquery in the correlated predicates), and the nested attributes should be the linked attribute plus the primary key of the relation in the subquery.

For convenience, we first formulate an auxiliary lemma about nest.

LEMMA 2 (NEST). *Let r be a relation over the relation schema R. Let $X \subseteq attr(R)$ and $Y \subseteq attr(R)$, and X and Y are disjoint. Then $\forall t \in \upsilon_{\{X\},R_1=\{Y\}}(r)$, there exists a unique $T \subseteq r$ such that $\forall t' \in T$, $t'[X] = t[X]$, $t'[Y] \in t[R_1]$, and $\forall t'' \in (r - T), t''[X] \neq t'[X]$.*

The following lemma shows that pushing down nest past join is equivalent to a join followed by a nest for one-level nested queries.

LEMMA 3 (PUSHING DOWN NEST PAST JOIN). *Let r and s be two relations over the relation schemas R and S, respectively. Let $A = B$ be a join condition between r and s, where $A \in attr(R)$, and $B \in attr(S)$. Let $r.\#$ denote the primary key of r. If $X \subseteq attr(R)$, $Y \subseteq attr(S)$, $r.\# \in X$, and $B \notin Y$, then $\upsilon_{\{X\},R_1=\{Y\}}(r \rtimes_{A=B} s) = \pi_{X,R_1}(r \rtimes_{A=B} (\upsilon_{\{B\},R_1=\{Y\}}(s)))$.*

Similarly, the nest operation can be pushed down past outer join.

LEMMA 4 (PUSHING DOWN NEST PAST OUTER JOIN). *Let r and s be two relations over the relation schemas R and S respectively. Let $A = B$ be a join condition between r and s, where $A \in attr(R)$, and $B \in attr(S)$. Let $r.\#$ denote the primary key of r. If $X \subseteq attr(R)$, $Y \subseteq attr(S)$, $r.\# \in X$, and $B \notin Y$, then $\upsilon_{\{X\},R_1=\{Y\}}(r \overrightarrow{\rtimes}_{A=B} s) = \pi_{X,R_1}(r \overrightarrow{\rtimes}_{A=B} (\upsilon_{\{B\},R_1=\{Y\}}(s)))$.*

One notable point is that using Lemma 3 or Lemma 4 may not always be efficient. For Lemma 3, assume that the size of the result of $r \rtimes_{A=B} s$ is M, and

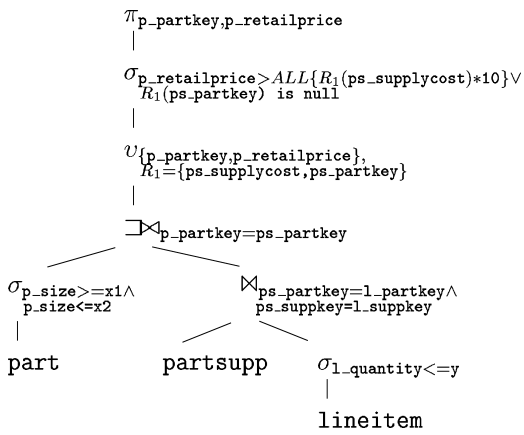


Fig. 4. Original Query Tree for Query 2.

the cost of $r \bowtie\!\!\!\triangleleft_{A=B} s$ is $Cost_{join}$, then the cost of LHS is $Cost_{join} + 2Mlog_{B-1}M$, where B is the available number of buffers in memory. For RHS, assume that the size of s is M_s, and the cost of a join on r and a nested relation obtained by nesting s is $Cost'_{join}$, then the cost of RHS is $Cost'_{join} + 2M_s log_{B-1}M_s$. When a nested query is evaluated using the nested relational approach, only hash joins are required. That means the relations in the outer query and the subquery need to be accessed only once. If we perform nest before join, the relation in the subquery has to be accessed at least twice: once for nest and once for join. Only when the size of the nested relation is small enough to significantly reduce the cost of join, that is $M_s < M$ and $Cost'_{join} < Cost_{join}$, such transformations may perform better. Thus, the key consideration of pushing down nest is the original size and the *reduced* size (by applying nest) of the relation in the subquery.

Example 5.1. To illustrate how to apply the above transformation rules on the nested relational approach, consider the following query.

Query 2 (*A One-Level Nested Query*). This query lists the parts, for a given size, whose retail prices are greater than all of 10 times of the supply cost for a specified line item quantity.

```
select p_partkey, p_retailprice
from   part
where  p_size >= x1 and p_size <= x2 and
       p_retailprice > all (select ps_supplycost * 10
                            from   partsupp, lineitem
                            where  p_partkey = ps_partkey and
                                   ps_partkey = l_partkey and
                                   ps_suppkey = l_suppkey and
                                   l_quantity <= y)
```

This query is derived from the TPC-H benchmark [Transaction Processing Performance Council]. It is a one-level nested query with a negative linking operator, ALL. Figure 4 shows the nested relational approach applied to Query 2.

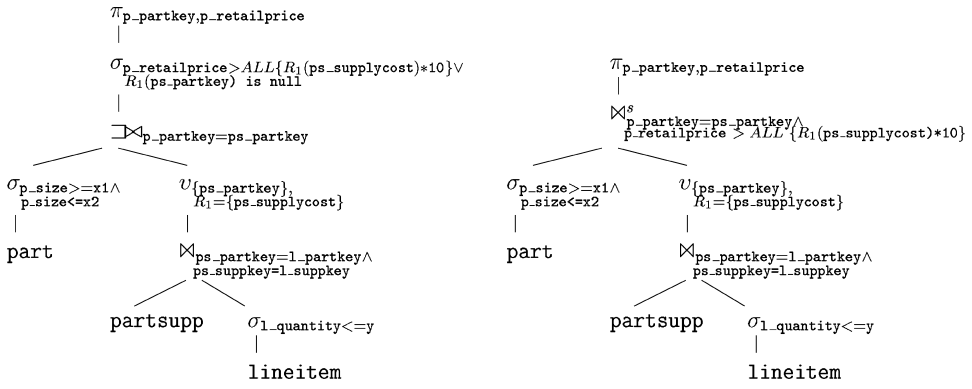


Fig. 5. Transformed Query Trees for Query 2.

Applying Lemma 3 to the query tree shown in Figure 4, Query 2 has an alternative plan shown in Figure 5(a).

5.3 Linking Selection and Join

After pushing down nest past join, one more possible optimization to the nested relational approach is to push down the linking selection into a join operation. It is not possible to use the traditional join operator, but it is feasible to use a join operator on set-valued attributes. Informally, joins on set-valued attributes, primarily *set containment join*, can be expressed as $r \rhd\lhd_{X \subseteq Y} s$, where r and s are nested relations, X and Y are set-valued attributes of relations r and s respectively. The result of $r \rhd\lhd_{X \subseteq Y} s$ is the combination of tuples $t_r \in r$ and $t_s \in s$ such that set $t_r.Y$ contains all elements in $t_s.X$. Several efficient algorithms and strategies for realizing this operator in an RDBMS have been proposed [Helmer and Moerkotte 1997; Ramasamy et al. 2000; Zhang et al. 2001; Melnik and Garcia-Molina 2002; Melnik and Garcia-Molina 2003; Mamoulis 2003]. The linking predicate is similar to set containment join in that it is a set-based predicate if we consider the linking attribute as a set with a singleton. However, the linking predicate is more complicated than set containment join in two aspects: first, the set computation involved in a linking predicate does not limit to set containment, it depends on the linking operator; second, for correlated subqueries, the linking predicate is computed for each group of the correlated attributes, rather than the whole result. Thus, we propose a generalization of set containment join that we call *generalized set join*, denoted by $\rhd\lhd^s$.

Definition 5.1. Let $R = (A_1, \ldots, A_m)$, $S = (B_1, \ldots, B_n, S_1, \ldots, S_p)$ be relation schemas, where $A_1, \ldots, A_m$ and $B_1, \ldots, B_n$ are atomic attributes, $S_1, \ldots, S_p$ are relation-valued attributes. Let r, s be relations on schemas R and S respectively. The generalized set join of r and s, $r \rhd\lhd^s_{A_i = B_j \wedge [A_k] \theta L \{S_l\}} s$, where $A_i, A_k \in \{A_1, \ldots, A_m\}$, $B_j \in \{B_1, \ldots, B_n\}$, $S_l \in \{S_1, \ldots, S_p\}$, θL is the

linking operator, and $[A_k]$ denotes A_k is optional. is defined as follows:

$$r \ltimes^{s}_{A_i=B_j \wedge [A_k]\theta L\{S_l\}} s := \{t \mid \exists t_r \in r \wedge t_s \in s \wedge t_r[A_i] = t_s[B_j] \wedge t_r[A_k]\,\theta\ L\ t_s[S_l]\texttt{is true} \wedge t[attr(R)] = t_r[attr(R)] \wedge t[attr(S)] = t_s[attr(S)]\}$$

The idea is to iterate over the groups defined by the attributes B_j. The linking predicate is computed for each group. Algorithms for realizing the set containment join operator (e.g., Mamoulis [2003]) can be slightly modified to implement our generalized set join operator.

LEMMA 5 (GENERALIZED SET JOIN). *Let r and s be two relations over the relation schemas R and S respectively. Let $A = B$ be a join condition between r and s, where $A \in attr(R)$, $B \in attr(S)$. Also, let r.# denote the primary key of r, and $[C]\theta L\{D\}$ denote a linking predicate, where the square bracket denotes optional. If $X \subseteq attr(R)$, $Y \subseteq attr(S)$, r.# $\in X$, $C \in X$ and $D \in Y$, then* $\sigma_{[C]\theta L\{R_1(D)\}}(\pi_{X,R_1}(r \ltimes_{A=B}(\upsilon_{\{B\},R_1=\{Y\}}(s)))) = \pi_{X,R_1}(r \ltimes^{s}_{A=B\wedge[C]\theta L\{R_1(D)\}} (\upsilon_{\{B\},R_1=\{Y\}}(s)))$.

In Lemma 5, we use the same $\ltimes^s$ for both positive and negative linking predicates. But for negative linking predicates, $\forall t_r \in r$, $\forall t^n_s \in \upsilon_{\{B\},R_1=\{Y\}}(s)$, if $t_r[A] = t^n_s[B]$ is false, which means that a tuple in the outer query has no matching tuples in the subquery (correlated predicates are not satisfied), t_r should be output as a part of result.

Example 5.2. By applying Lemma 5, the query tree shown in Figure 5(a) can be transformed to the query tree shown in Figure 5(b).

6. OPTIMIZATION OF SPECIAL QUERIES

The nested relational approach could be further optimized for some special queries to gain better performance. The nested relational approach always unnests subqueries first. However, it may not be efficient for some special linking operators and special queries. In this section, we propose several transformation rules for some special linking operators to avoid materialization. We also propose alternative solutions for *linear correlated queries* and *tree queries with redundancy* for better performance.

6.1 Queries with Positive Linking Operators

Although the nested relational approach is focused on dealing efficiently with mixed and negative linking operators, we would like the approach to be also efficient for positive linking operators. However, existing approaches have a very efficient plan for evaluating positive linking operators. In the case of `IN`, for instance, the linking predicate is transformed into a semijoin, while the nested relational approach would create a join, a nest and a linking selection. The following lemmas will show that through algebraic rewriting, the nested relational approach is equivalent to the standard existing one for positive cases.

LEMMA 6 (ANY $\Longrightarrow$ SEMIJOIN). *Let r and s be two relations over the relation schemas R and S respectively. Let $A\theta_1 B$ be a join condition between r and s, where $A \in attr(R)$, $B \in attr(S)$, and $\theta_1 \in \{<, \leq, >, \geq, =, \neq\}$. Also, let $r.\#$ denote the primary key of r. If $X \subseteq attr(R)$, $Y \subseteq attr(S)$, $r.\# \in X$, and $L \subseteq X$, then $\pi_L(\sigma_{C\theta_2 ANY\{R_1(D)\}}(\upsilon_{\{X\},R_1=\{Y\}}(r \bowtie\!\!\!\triangleright_{A\theta_1 B} s))) = \pi_L(r \ltimes_{A\theta_1 B \wedge C\theta_2 D} s)$, where $C \in X$, $D \in Y$, $\theta_2 \in \{<, \leq, >, \geq, =, \neq\}$, and $\ltimes$ denotes semijoin.*

LEMMA 7 (EXISTS $\Longrightarrow$ SEMIJOIN). *Let r and s be two relations over the relation schemas R and S respectively. Let $A\theta B$ be a join condition between r and s, where $A \in attr(R)$, $B \in attr(S)$ and $\theta \in \{<, \leq, >, \geq, =, \neq\}$. Also, let $r.\#$ denote the primary key of a relation r. If $X \subseteq attr(R)$, $Y \subseteq attr(S)$, $r.\# \in X$, $s.\# \in Y$, and $L \subseteq X$, then $\pi_L(\sigma_{EXISTS\ \{R_1(Y)\}}(\upsilon_{\{X\},R_1=\{Y\}}(r \bowtie\!\!\!\triangleright_{A\theta B} s))) = \pi_L(r \ltimes_{A\theta B} s)$, where $\ltimes$ denotes semijoin.*

Note that Lemma 6 and Lemma 7 are proved based on one-level nested queries. Since only the attributes in one relation that participate in a semijoin are kept, for multi-level queries, semijoin might be replaced by join to keep attributes in both relations required for later processing.

6.2 One-Level Query with NOT EXISTS Linking Operator

Unlike `ALL` or `NOT IN` linking predicates, `NOT EXISTS` linking predicates are not affected by null values. The result of a `NOT EXISTS` linking predicate only depends on whether the subquery result is empty or not. Similar to simplifying an `EXISTS` linking predicate to a semijoin, a `NOT EXISTS` linking predicate in a one-level nested query can be reduced to an antijoin of the relations in the outer query and the subquery.

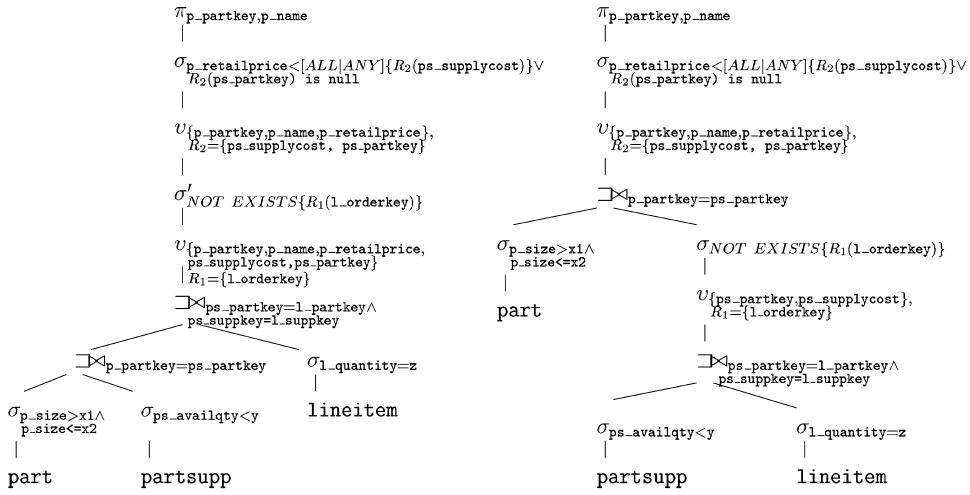
LEMMA 8 (NOT EXISTS $\Longrightarrow$ ANTIJOIN). *Let r and s be two relations over the relation schemas R and S respectively. Let $A\theta B$ be a join condition between r and s, where $A \in attr(R)$, $B \in attr(S)$ and $\theta \in \{<, \leq, >, \geq, =, \neq\}$. Also, let $r.\#$ denote the primary key of a relation r. If $X \subseteq attr(R)$, $Y \subseteq attr(S)$, $r.\# \in X$, $s.\# \in Y$, and $L \subseteq X$, then $\pi_L(\sigma_{NOT\ EXISTS\ \{R_1(Y)\}}(\upsilon_{\{X\},R_1=\{Y\}}(r \bowtie\!\!\!\square_{A\theta B} s))) = \pi_L(r \bar{\ltimes}_{A\theta B} s)$, where $\bar{\ltimes}$ denotes antijoin.*

Note that Lemma 8 can be used only for a one-level query with a `NOT EXISTS` linking predicate. For multi-level nested queries, a `NOT EXISTS` linking predicate has to be evaluated using outerjoin, nest, and linking selection unless a part of the query can be considered as a one-level nested query (an example will be shown below).

6.3 Linear Correlated Queries

A *linear correlated query* denotes the nested query in which each inner query block is only correlated to its adjacent outer query block. Since evaluation of the inner query block only depends on its adjacent outer query block, linear correlated queries can be processed efficiently from bottom-up instead of from top-down.

Example 6.1. The following query is a linear correlated query because the third query block is only correlated to the second query block, and the



(a) Top-Down (b) Bottom-Up

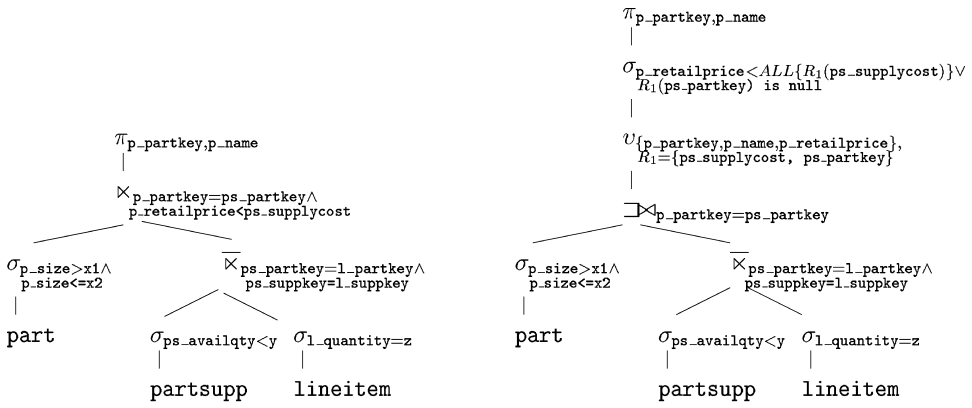
Fig. 6. Original Query Trees for Query 3.

second is only correlated to the first. The term [any|all] refers to choosing either one.

Query 3 (*A Linear Correlated Query*). This query lists the parts, for a given size, whose retail prices are less than any or all of the supply costs for a specified available quantity in which no lineitem with a given quantity exists.

```
select p_partkey, p_name
from   part
where  p_size >= x1 and p_size <= x2 and
       p_retailprice < [any|all]
         (select ps_supplycost
          from   partsupp
          where  ps_partkey = p_partkey and ps_availqty < y and
                 not exists (select *
                             from   lineitem
                             where  ps_partkey = l_partkey and
                                    ps_suppkey = l_suppkey and
                                    l_quantity = z))
```

The query tree by applying the nested relational approach from top-down on Query 3 is shown in Figure 6(a). Due to its linear correlated feature, there is an alternative to the nested relational approach: evaluating the third query block first, and then the second. Thus, the second and the third query blocks form a one-level nested query with a NOT EXISTS linking operator. Later on, the first and the second query blocks form another one-level nested query with an ALL linking operator. Both of these two one-level nested queries can be evaluated using the nested relational approach. The query tree by applying the nested relational approach from bottom-up on Query 3 is shown in Figure 6(b). Here



(a) Query 3a(mixed: ANY/NOT EXISTS)

(b) Query 3b(negative: ALL/NOT EXISTS)

Fig. 7. Transformed Query Trees for Query 3.

we consider two variations of Query 3. The first one is Query 3a with mixed ANY and NOT EXISTS operators. The query tree shown in Figure 6(b) can be simplified to the query tree shown in Figure 7(a) by applying Lemma 6 and Lemma 8. In other words, Query 3a can be evaluated by an antijoin followed by a semijoin. The second variation is Query 3b with two negative ALL and NOT EXISTS operators. In this case, only the NOT EXISTS linking predicate can be simplified to an antijoin, the ALL linking predicate has to be evaluated by the original way. The query tree is shown in Figure 7(b). Clearly, the bottom-up strategy might avoid (outer) join operations on large relations.

Although an overall, uniform approach is very important, it is also important in an optimization effort to point out possibilities for refinement. We point out linear queries as a special case that offers further opportunities for optimization. Basically, since linear correlated queries can be evaluated (using the nested relational approach) either from top-down or from bottom-up, but generally the bottom-up strategy may have more efficient performance (because it can avoid large join operations). In the general case what we do is to identify any blocks of linear correlate subqueries. If there are some query blocks are linear correlated, and some are not, we break sets of adjacent blocks into parts with all linear correlated and all not linear correlated. For linear correlated part, we can follow the bottom-up strategy, and for non linear correlated part we follow the top-down strategy. After computing each part, we identify the relationship between adjacent parts: is it linear correlated or not? If yes, we use the bottom-up strategy; otherwise, we use top-down strategy. We follow this procedure recursively until all parts are solved.

6.4 Tree Queries with Redundancy

We say a nested query with redundancy means that the FROM and WHERE clauses in the outer query and the subquery or among subqueries share common tables

and conditions. It is important to point out that redundancy is present because of the structure of SQL, which necessitates a subquery in order to declaratively state an attribute to be computed. The redundancy problem has been studied by several researchers [Rao and Ross 1998; Roy et al. 2000; Galindo-Legaria and Joshi 2001; Zuzarte et al. 2003]. All these approaches focus on the redundancy appearing in the outer query and the subquery. Here we consider the redundancy present among subqueries. Specifically, we are concerned with a nested tree query in which at a certain level, the tables in the subqueries are the same and the correlated predicates are the same. As an example, consider the following query:

Query 4 (*A Tree Query with Redundancy*). This query lists the suppliers, for a given nation and a given order status, whose product was a part of a multi-supplier order and extended prices are less than all extended prices for a specified lineitem quantity, where they were the only supplier who failed to meet the committed delivery date.

```
select s_name
from   supplier, lineitem l1, orders, nation
where  s_suppkey = l1.l_suppkey and o_orderkey = l1.l_orderkey and
       o_orderstatus = 'F' and l1.l_receiptdate > l1.l_commitdate and
       s_nationkey = n_nationkey and n_name = 'SAUDI ARABIA' and
       l_extendedprice < all
                  (select l_extendedprice
                   from   lineitem l2
                   where  l2.l_orderkey = l1.l_orderkey and
                          l2.l_suppkey <> l1.l_suppkey and
                          l2.l_quantity = 10) and
       not exists (select *
                   from   lineitem l3
                   where  l3.l_orderkey = l1.l_orderkey and
                          l3.l_suppkey <> l1.l_suppkey and
                          l3.l_receiptdate > l3.l_commitdate and
                          l3.l_quantity = 10)
```

Query 4 is a one-level nested tree query with two subqueries at the same level. The two subqueries have the same table `lineitem` and the same correlated predicates. In the nested relational approach, a nested tree query is evaluated by choosing an arbitrary subquery (perhaps the least expensive one) to evaluate first and then the resulting tuples can be used to evaluate another subquery. In other words, The tables in the subqueries are accessed as many times as the number of subqueries. The query tree by applying the nested relational approach is shown in Figure 8. In accordance with Lemma 8, it can be optimized by using antijoin for the `NOT EXISTS` linking operator. Suppose we evaluate the `NOT EXISTS` subquery first, the query tree is shown in Figure 9(a). Even though antijoin is the most efficient strategy for `NOT EXISTS`, in our example query, the table `lineitem` still need to be accessed twice.

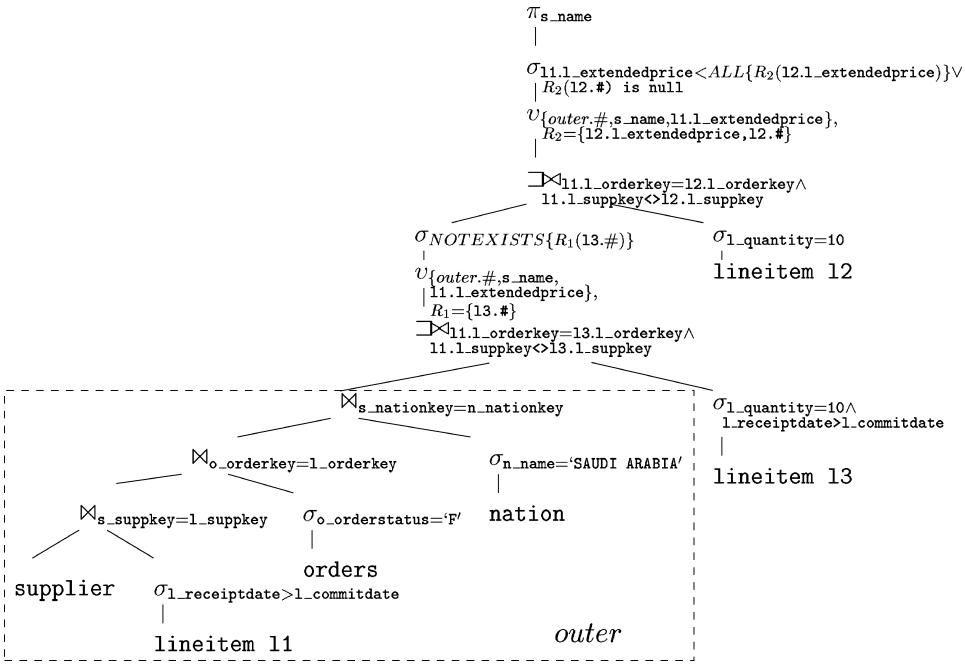


Fig. 8. Original Query Tree for Query 4.

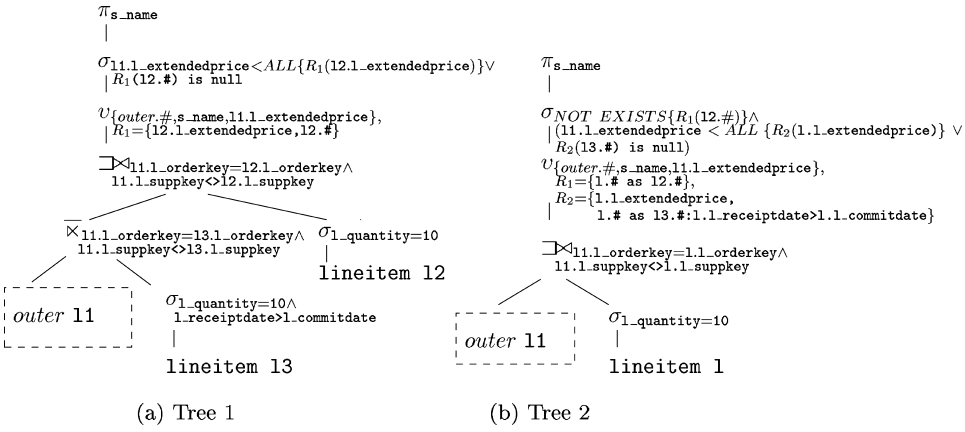


Fig. 9. Transformed Query Trees for Query 4.

Is there an alternative to outperform this strategy? Clearly, if `lineitem` can be accessed only once when computing the two subqueries, the performance must be better. To achieve this goal, we need an operator to allow several subqueries to be computed simultaneously based on different conditions.

The nest operator defined in Definition 3.6 has a set of nesting attributes and a set of nested attributes; furthermore, nested attributes depend on nesting attributes. Under some circumstance, for a set of nesting attributes, the nest operator might require more than one sets of nested attributes which are

independent of each other; even more complicated, the set of nested attributes might be based on different conditions. This leads us to introduce a more generalized, conditional nest operator.

Definition 6.1. Let r be a nested relation over the nested schema R. Let $N_1, N_2, \ldots, N_m$ be m disjoint subsets of $attr(R)$. Let $C_2, \ldots, C_m$ be conditions over $N_2, \ldots, N_m$. Then the *conditional nest* of r by N_1 keeping N_2 with the condition $C_2, \ldots$, and keeping N_m with the condition C_m, $\upsilon_{N_1,R_1=N_2:C_2,\ldots,R_m=N_m:C_m}(r)$, where $R_1, \ldots, R_m$ are renames of $N_2, \ldots, N_m$, is defined as:

$$\begin{aligned}\upsilon_{N_1,R_1=N_2:C_2,\ldots,R_m=N_m:C_m}(r) := \{t \mid \exists t' \in r \wedge\ & t[N_1] = t'[N_1] \wedge \\ & t[R_1] = \{t''[N_2] \mid t'' \in r \wedge \\ & \qquad t''[N_1] = t[N_1] \wedge C_2(t'') \texttt{ is true}\} \wedge \\ & \vdots \\ & t[R_m] = \{t''[N_m] \mid t'' \in r \wedge \\ & \qquad t''[N_1] = t[N_1] \wedge C_m(t'') \texttt{ is true}\}\}\end{aligned}$$

Note that null values in N_1 are considered the same as defined in Definition 3.6.

Given a one-level tree query with two subqueries, for the outer query block, let r denote the tables in the `FROM` clause, C denote the conditions in the `WHERE` clause, A_{lg1} denote the first linking attribute and A_{lg2} denote the second linking attribute. For the first subquery and the second subquery, let s denote the tables in the `FROM` clause, A_{ld1} and A_{ld2} denote the linked attributes respectively, and C_1 and C_2 denote the conditions in the `WHERE` clause of the first subquery and the second subquery respectively. Assume the correlated predicates are the same. Let *attr* denote the attributes in the `FROM` clause of the first query block. Then the query can be evaluated as follows: First, we perform a (outer) join of $\sigma_C(r)$ and $\sigma_{C_1 \cap C_2}(s)$ on the correlated predicates. Second, we use the generalized nest operator with the form of $\upsilon_{\{r.\#,A_{lg1},A_{lg2},attr,\},R_1=\{A_{ld1}:C_1-C_2\},R_2=\{A_{ld2}:C_2-C_1\}}$ to nest the result of join. Third, we compute two linking predicates simultaneously. Finally, we project the required attributes.

The query tree using the generalized nest is shown in Figure 9(b). Clearly, `lineitem` needs to be accessed only once.

The nested relational algebra defined in Section 3 is complete in the sense that it includes all operators required by the nested relational approach. We define the conditional nest operator for optimization purposes because it gives a concise expression and an efficient implementation. The conditional operator is only syntactic sugar, in the sense that it can be expressed by the operators introduced in Section 3. Let r be a flat relation with the attributes A, B, C. To obtain a nested relation with the schema $R = (A, (B), (C))$, where A is the nesting attribute, B and C are nested attributes and they are independent of each other, we need a series of expressions that use the operators defined in Section 3: $T_1 = \pi_{A,B}(r)$, $T_2 = \pi_{A,C}(r)$, $T_3 = \upsilon_{\{A\},\{B\}}(T_1)$, $T_4 = \upsilon_{\{A\},\{C\}}(T_2)$, The final nested relation can be obtained by $T_3 \bowtie_{T_3.A=T_4.A} T_4$. This has been proved by Van Gucht and Fischer [1986]. However, the expression using the conditional nest operator is: $\upsilon_{A,R_1=\{B\},R_2=\{C\}}(r)$. Clearly, it is simpler than the

previous expressions, and most importantly, it can be implemented efficiently. If we use the series of expressions, we need to scan r twice for projecting A, B and A, C respectively, then sort the projected relations to make nested relations, and finally perform joins on the nested relations. However, for the expression using the conditional nest operator, it can be implemented efficiently by sorting r on A once and simultaneously making the group on B and C.

7. EXPERIMENTS AND PERFORMANCE ANALYSIS

To verify the efficiency of the nested relational approach, we have implemented it on top of a leading commercial DBMS, which we call "System A" in this article. In this section, we compare the performance of the nested relational approach to the native approach of System A in evaluating queries containing nonaggregate subqueries, as well as the performance among different implementations of the nested relational approach.

7.1 Implementations

To implement the nested relational approach, we wrote stored procedures in *procedural SQL*, an extension of SQL that adds programming-language-like capabilities (variable declaration, loop, conditional statements, etc.) to SQL. The nested relational approach is designed in two stages: first, an SQL query is used to unnest the query by executing (outer) joins of the base relations in each query block with corresponding selections pushed down. Second, code in the procedure implements the nest operator and the linking selection operator by processing the tuples fetched from the first stage, which we call the *intermediate result*. We implemented Naive NRA, Nest-Merged NRA (for multi-level queries), and Pipelined NRA. In order to simulate the nest operator in an effective manner, we make the database sort the intermediate result in the first step. For simplicity, we store subrelations inline with other attributes. The reasons to use stored procedures to implement the nested relational approach are: first, they run inside the database so that communication overhead can be reduced significantly compared to external processing; second, they can be called by other applications, which makes the nested relational approach more suited for practical use. However, there still exists communication overhead when the stored procedure fetches data from the SQL engine (as observed by Akinde and Bohlen [2003], this is one considerable disadvantage that all experimental settings similar to ours must bear).

7.2 Experiment Setup

In our experiments, we created a TPC-H database [Transaction Processing Performance Council] at scale factor 1 (total data size 1GB) in System A, hosted on a server with an Intel Pentium 4 2.80GHz processor, two 36GB SCSI disks, and 1GB memory, running Red Hat Enterprise Linux WS release 3. We configured a buffer cache of size 32MB, and installed all data and indexes in a single disk. Indexes on the primary key of each base table were automatically built by System A. Additional indexes on the selected foreign keys were created manually when needed.

7.3 Performance Analysis

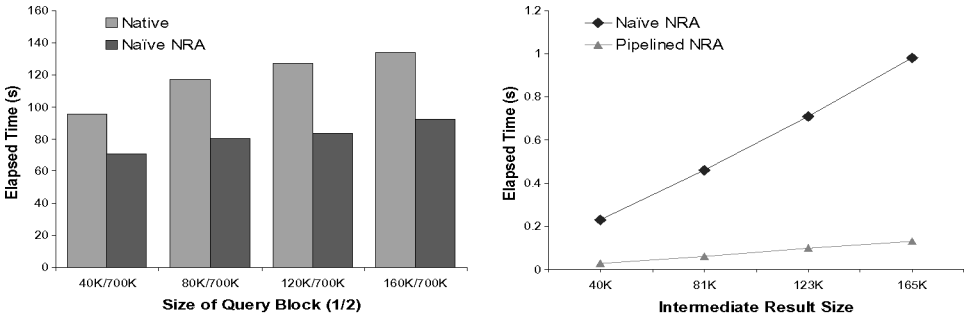
Five queries and their variations with four different sizes derived from the TPC-H benchmark were tested in our experiments. The performance metric is the elapsed time of running each query. We take the average time of three independent runs for each query, which does not include the time for the first run. Before each running, the buffer cache of System A is flushed. In reporting our experiment results, we have two kinds of charts: charts with the title "Native vs. Naive NRA" and charts with the title "Nest&Linking Selection". The charts titled "Native vs. Naive NRA" denote the comparison between the native approach of System A and Naive NRA, and plot the elapsed time of query running on the Y-axis and the size of each reduced query block (outer/inner) on the X-axis. The size of each reduced query block denotes the size of the base table (or joins of base tables) in a query block with corresponding selections pushed down, but without linking predicates executed yet. We choose this size as a parameter due to the fact that it directly relates to the intermediate result, which in turn, relates to the overhead corresponding to fetching tuples from the SQL engine. This size is controlled by changing constants on the selections and thus varying their selectivity factor. The charts titled "Nest&Linking Selection" denote the comparison among Naive NRA, Nest-Merged NRA (for multi-level queries), and Pipelined NRA, and plot the elapsed time of processing the nest operator and the linking selection operator on the Y-axis and the size of the intermediate result on the X-axis. Detailed performance analysis can be found in the electronic appendix.

7.3.1 *A One-Level Query with a Negative Linking Operator.* Our first experiment was done on Query 5, which is a one-level nested query with an `ALL` linking operator.

Query 5 (*A One-Level Query with a Negative Linking Operator*). This query lists the orders placed on a given date whose total prices are greater than the extended price of the line item shipped, committed, and received in a given order.

```
select o_orderkey, o_orderpriority
from   orders
where  o_orderdate >= x1 and o_orderdate < x2 and
       o_totalprice > all (select l_extendedprice
                           from   lineitem
                           where  l_orderkey = o_orderkey and
                                  l_commitdate < l_receiptdate and
                                  l_shipdate < l_commitdate)
```

The native approach evaluates Query 5 in the nested iteration manner with the use of index rowid on `lineitem`. We implemented both Naive NRA and Pipelined NRA. We can see that Naive NRA outperforms the native approach, although the native approach benefits from indexes (see Figure 10(a)), and Pipelined NRA is more efficient than Naive NRA (see Figure 10(b)).



(a) Native vs. Naive NRA (b) Nest&Linking Selection

Fig. 10. Query 5.

7.3.2 *Two-Level Queries with Mixed, Negative, Positive Linking Operators.* Our second experiment was performed on three variations of Query 6, which is a two-level nested query with three query blocks. From top-down, the second query block is correlated to the first query block and the third query block is correlated to both the other two query blocks. The terms [`all|any`], [`exists|not exists`] and [= | <>] denote choosing either one.

Query 6 (*A Two-Level Query with Different Linking Operators*). This query answers the similar question as Query 3.

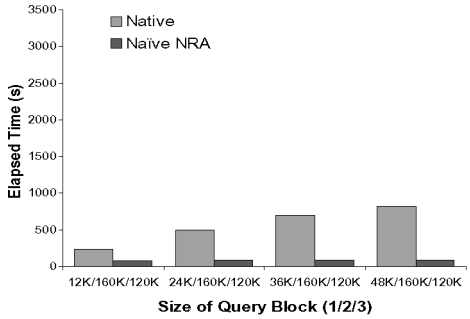
```
select p_partkey, p_name
from   part
where  p_size >= x1 and p_size <= x2 and
       p_retailprice < [all|any]
         (select ps_supplycost
          from   partsupp
          where  ps_partkey = p_partkey and
                 ps_availqty < y and [exists|not exists]
                   (select *
                    from   lineitem
                    where  p_partkey [=|<>] l_partkey and
                           ps_suppkey [=|<>] l_suppkey and
                           l_quantity = z))
```

Three variations of Query 6 are used to test mixed linking operators, negative linking operators, and positive linking operators:

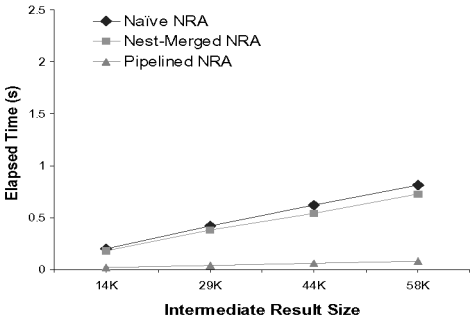
(1) *Query* 6a: Mixed linking operators `ALL` and `EXISTS`.
(2) *Query* 6b: Two negative linking operators `ALL` and `NOT EXISTS`.
(3) *Query* 6c: Two positive linking operators `ANY` and `EXISTS`.

Generally, query optimizer generates query plans depending on not only linking operators but also correlated predicates. Thus, each variation again has three cases based on the correlated predicates in the third query block:

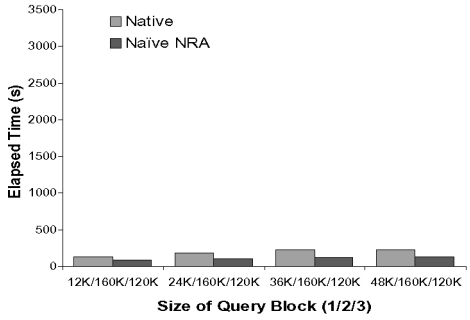
(I) `p_partkey = l_partkey and ps_suppkey = l_suppkey`.



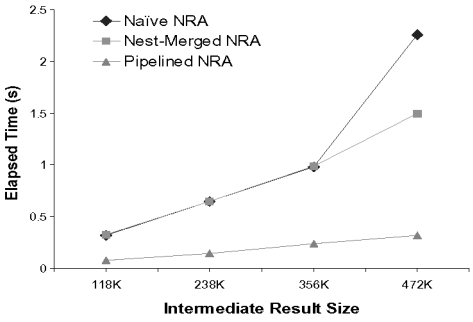
(a) Query 6a(I): Native vs. Naive NRA



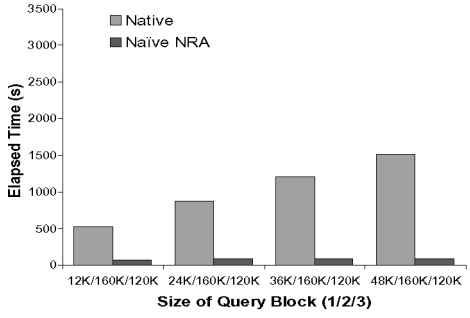
(b) Query 6a(I): Nest&Linking Selection



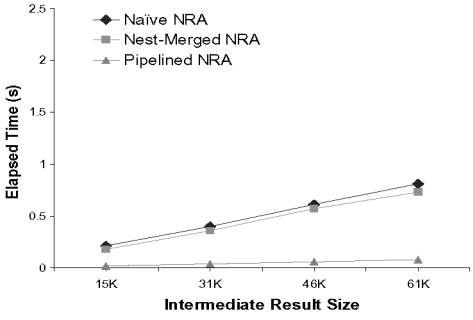
(c) Query 6a(II): Native vs. Naive NRA



(d) Query 6a(II): Nest&Linking Selection



(e) Query 6a(III): Native vs. Naive NRA

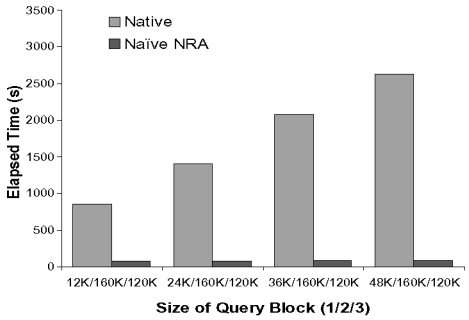


(f) Query 6a(III): Nest&Linking Selection

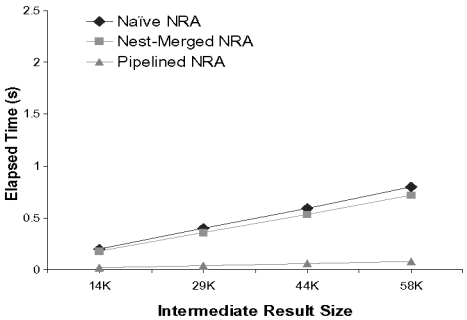
Fig. 11. Query 6a(mixed: `ALL`/`EXISTS`).

(II) `p_partkey <> l_partkey and ps_suppkey = l_suppkey.`
(III) `p_partkey = l_partkey and ps_suppkey <> l_suppkey.`

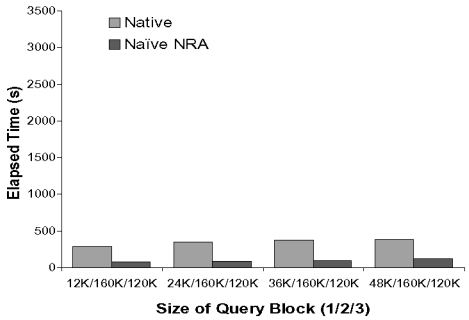
It is important to point out that *System A is unable to use antijoin in these queries, even though the* `NOT NULL` *constraint is added*; this is due to the problems mentioned in Section 2. System A has different plans for all test queries. For Query 6a (see Figure 11) and Query 6c (see Figure 13), System A always tries to unnest the third query block due to the `EXISTS` linking predicate, while for



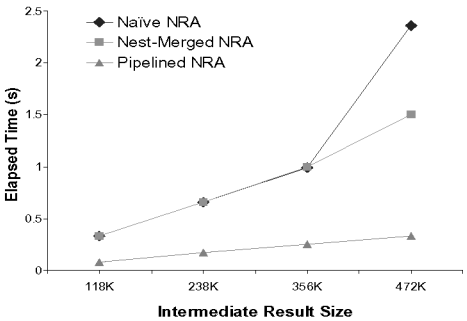
(a) Query 6b(I): Native vs. Naive NRA



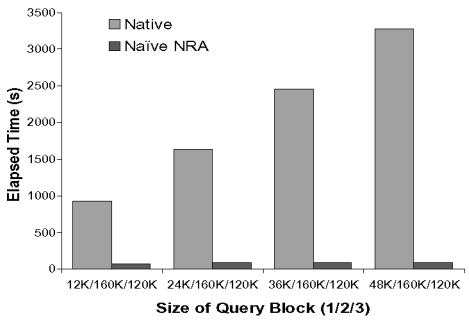
(b) Query 6b(I): Nest&Linking Selection



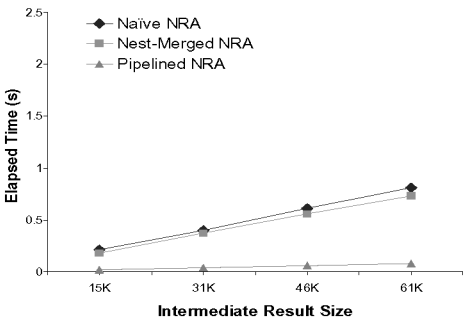
(c) Query 6b(II): Native vs. Naive NRA



(d) Query 6b(II): Nest&Linking Selection



(e) Query 6b(III): Native vs. Naive NRA

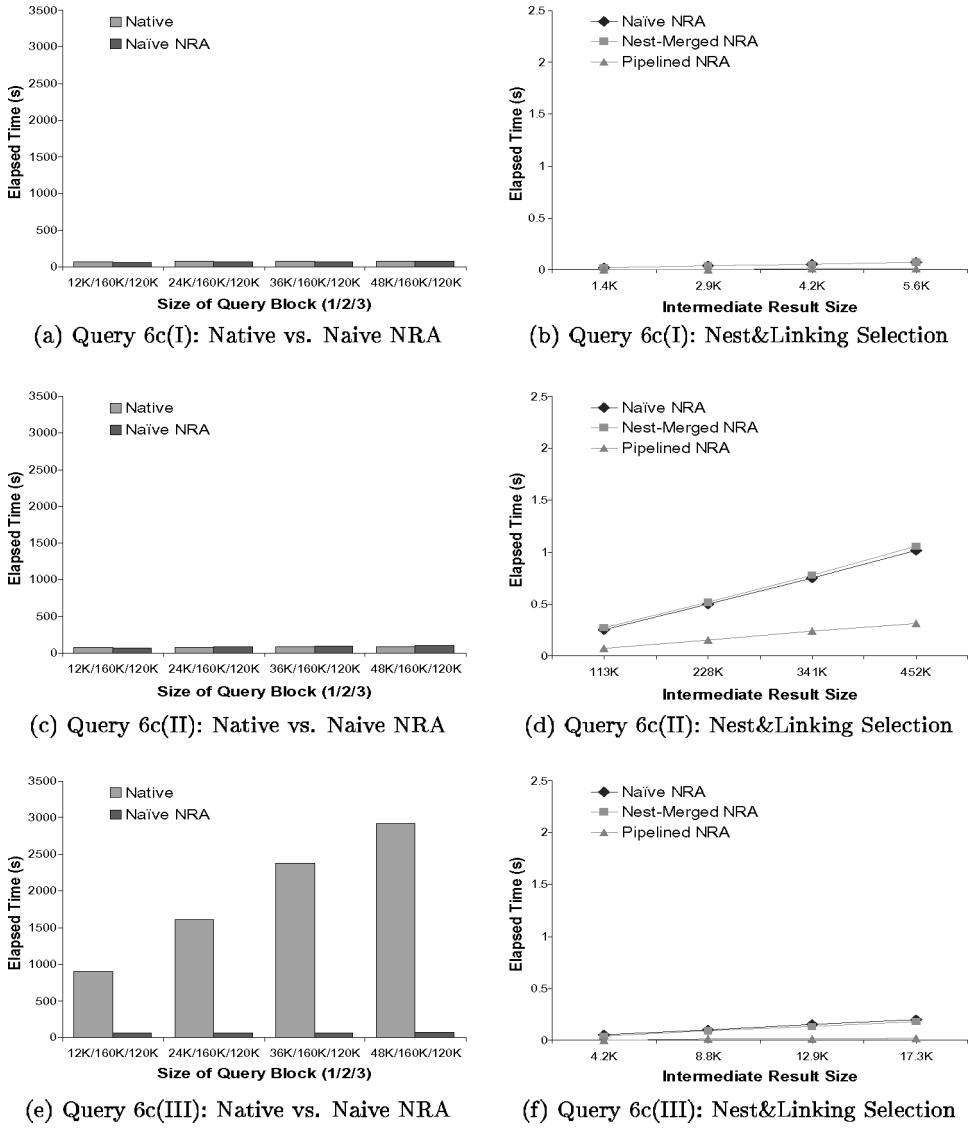


(f) Query 6b(III): Nest&Linking Selection

Fig. 12. Query 6b(negative: `ALL/NOT EXISTS`).

Query 6b (see Figure 12), System A has to perform nested iteration over three query blocks. Also, System A treats different correlated predicates in a different manner. More importantly, System A heavily depends on indexes.

Unlike System A, whose performance varies among similar queries, the nested relational approach performs almost equally among similar queries (see Figure 11, Figure 12, and Figure 13). We implemented Naive NRA, Nest-Merged NRA, and Pipelined NRA. Pipelined NRA always performs best among three approaches. Nest-Merged NRA performs just slightly better than Naive



(a) Query 6c(I): Native vs. Naive NRA

(b) Query 6c(I): Nest&Linking Selection

(c) Query 6c(II): Native vs. Naive NRA

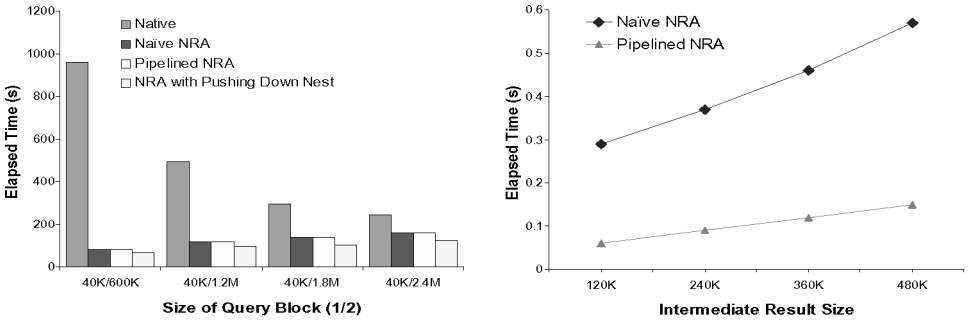
(d) Query 6c(II): Nest&Linking Selection

(e) Query 6c(III): Native vs. Naive NRA

(f) Query 6c(III): Nest&Linking Selection

Fig. 13. Query 6c(positive: `ANY/EXISTS`).

NRA if the size of the intermediate is not too large; with the increasing sizes of the intermediate result, Nest-Merged NRA shows the potential to outperform Naive NRA (see Figures 11(d) and Figure 12(d)). We can see the much better performance of Naive NRA compared to the native approach in both Figures 11 and 12. Naive NRA performs comparably to the native approach for Query 6c(I) (see Figure 13(a)) because of the extremely small size of the intermediate result, and significantly better than the native approach (using the single index on `l_partkey`) for Query 6c(III) (see Figure 13(e)), and slightly worse than the



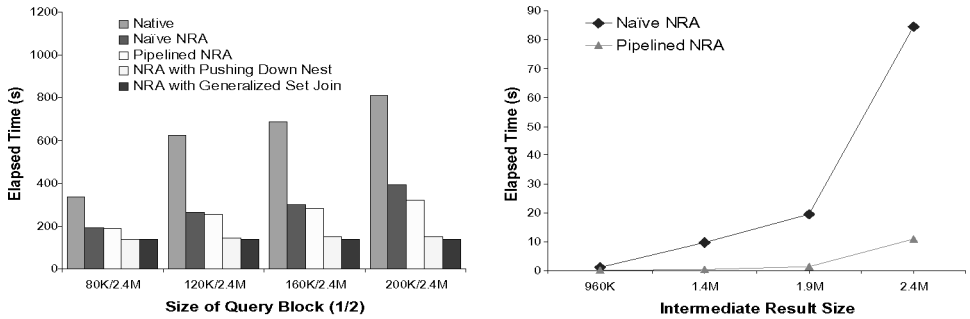
(a) Native vs. NRA (b) Nest&Linking Selection

Fig. 14. Query 2a.

native approach for Query 6c(II) (see Figure 13(c)) due to the large size of the intermediate result and communication overhead.

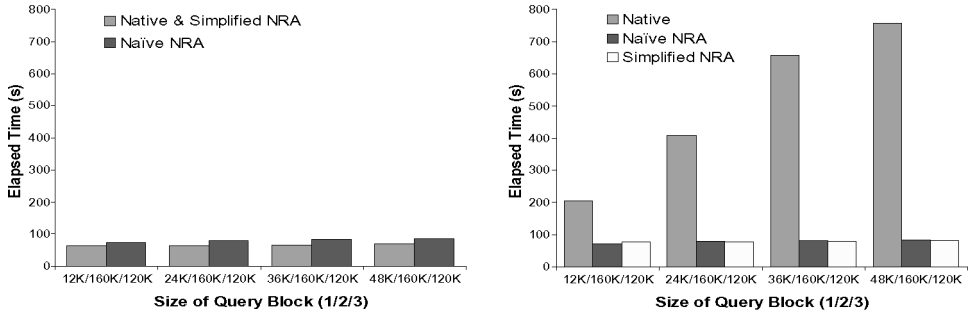
7.3.3 *Algebraic Optimizations.* Our third experiment is to study the algebraic optimization of the nested relational approach. Specifically, we consider two issues: pushing down nest past join (Lemma 3) and using generalized set join (Lemma 5). We performed an experiment on a variation of Query 2 (see Example 5.1), Query 2a. The size of the relation in the outer query is stable (40K tuples) and the size of the relation in the subquery ranges from 600K tuples to 2.4M tuples. We compare the performance of the native approach, Naive NRA, Pipelined NRA, and NRA with pushing down nest. The larger the ratio of the original size and the reduced size, the better the performance (see Figure 14(a)). Naive NRA and Pipelined NRA perform comparably (see Figure 14(b)). The comparison between the native approach and Naive NRA is similar to the previous experiments. An interesting point is that the native approach takes shorter time with the increasing sizes of the relation in the subquery.

With pushing down nest, one possible optimization is to use *generalized set join* (defined in Section 5.3) instead of a join followed by a linking selection. Our experiment was performed on another variation of Query 2 (see Example 5.1), Query 2b. The size of the relation in the subquery is 2.4M tuples and the size of the relation in the outer query ranges from 80K tuples to 200K tuples. We implemented the generalized set join in a sort-merge join manner using stored procedures. We compare the performance of the native approach, Naive NRA, Pipelined NRA, NRA with pushing down nest only, and NRA with generalized set join. Comparing NRA with pushing down nest only and NRA with generalized set join (see Figure 15(a)), we can see that the difference between them becomes more significant with the increasing size of the relation in the outer query. Furthermore, NRA with pushing down nest only and NRA with generalized set join perform much better than Naive NRA and Pipelined NRA, because the size of the nested relation is about 12 times smaller than the size of the original relation in the subquery. Another observation from Figure 15(a) is that the difference between Pipelined NRA and Naive NRA is more obvious than that



(a) Native vs. NRA (b) Nest&Linking Selection

Fig. 15. Query 2b.



(a) Query 3a(mixed: `ANY/NOT EXISTS`) (b) Query 3b(negative: `ALL/NOT EXISTS`)

Fig. 16. Query 3.

in the previous experiments. This is due to the large size of the intermediate result (see Figure 15(b), the maximum size of the intermediate result is 2.4M tuples). Clearly, the larger size of the intermediate result, the more efficient Pipelined NRA.

7.3.4 *Special Case I: Linear Correlated Queries.* The forth experiment we did is on two variations of Query 3 (see Example 6.1), a two-level linear correlated query.

Our first variation of Query 3 is Query 3a with mixed `ANY` and `NOT EXISTS` operators (see Figure 16(a)). The native approach evaluates Query 3a from bottom-up by antijoin first and then semijoin. Our second variation of Query 3 is Query 3b with two negative operators `ALL` and `NOT EXISTS` (see Figure 16(b)). The native approach can only unnest the `NOT EXISTS` linking predicate by antijoin, and perform nested iteration for the `ALL` linking predicate. We implemented Naive NRA on both queries. We also implemented Simplified NRA on Query 3b based on the query tree shown in Figure 7(b). Comparing Figure 16(a) and Figure 16(b), we can see that the nested relational approach has similar performance on nested linear queries regardless of the linking operators; the performance of the native approach depends on the existence of the `ALL` or `NOT IN` linking operator: the native approach performs significantly worse than

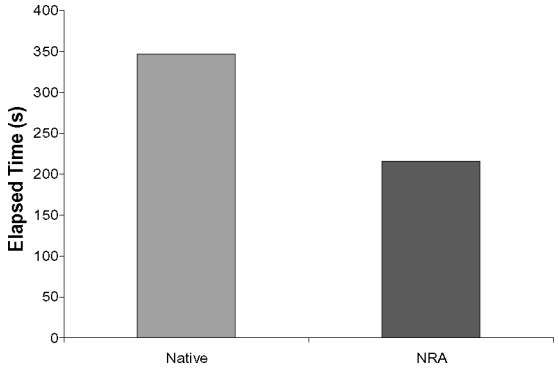


Fig. 17. Query 4.

Naive NRA if `ALL` or `NOT IN` linking operators are used (see Figure 16(b)), but slightly better than Naive NRA when `ALL` or `NOT IN` linking operators are not used (see Figure 16(a)).

7.3.5 *Special Case II: Tree Queries with Redundancy.* Our final experiment evaluated a tree query with redundancy using the native approach and the nested relational approach. We test Query 4 in Section 6.4, which has two nonaggregate subqueries with the linking operators `ALL` and `NOT EXISTS`. The native approach uses antijoin to compute the `NOT EXISTS` subquery (the most efficient plan among existing approaches), and the resulting tuples are used to compute the `ALL` subquery by nested iteration. In total, `lineitem` in the `NOT EXISTS` subquery is accessed once, `lineitem` in the `ALL` subquery is accessed using index every time when the subquery is invoked. By using the conditional nest operator, the nested relational approach requires only one access to `lineitem`. The experiment results shown in Figure 17 demonstrate a significant improvement of the nested relational approach compared to the native approach.

7.4 Discussion

To test the efficiency and stability of the nested relational approach, we also tried two multi-level queries, Query 3a and Query 6a(I) in another well-known commercial DBMS. Performances are very similar to those of System A. We believe the results validate our viewpoint, which can be summarized as follows: the traditional approach does not treat positive and negative linking predicates equally; it has problems unnesting negative linking predicates, and these problems get worse when the query complexity (especially number of levels) grows. By contrast, the cost of computing positive and negative linking predicates in the nested relational approach is about the same, as they are treated the same way. Also, as query complexity grows, the nested relational approach handles the query with little degradation in performance.

There is no single approach that works efficiently for all kinds of queries. The nested relational approach also exhibits limitations and trade-offs. Generally, the nested relational approach requires (outer) join operations to unnest

subqueries, and this intermediate result needs to be materialized for later processing. Such results may be large. Thus, the size of the intermediate result has a significant impact on the performance of the nested relational approach. Indeed, this is the main reason why we test four different sizes of intermediate results for each query. With the increasing size of the intermediate result, processing time increases polynomially. As query complexity grows, the nested relational approach only shows little degradation in performance. Also, we point out that: (a) the nested relational approach is a magic-like approach that applies all selections early within each level, thus helping out somewhat; (b) in some cases (top-down approach for linear queries, pushing down nest past join), it is not necessary to materialize all levels of a query at once; we can proceed step-by-step and alleviate the problem of large intermediate results; (c) we provide three different implementations in which Pipelined NRA may slightly improve the performance with one scan of the intermediate result; (d) even if the intermediate result is large, the nested relational approach may still be better than performing nested iteration over subqueries, especially when no indexes are present, as our experiments show. When indexes are present, nested iteration may be a good choice if the intermediate result is too large. We stress that we do not assume that the nested relational approach is always the best; we assume a cost-based approach: the optimizer can estimate the costs of each possible plan (including ours) and chooses the one with the least cost.

It is worth noting that for all test queries, except Query 6c(I) and Query 6c(II), the native approach takes full advantage of indexes. The nested relational approach is not affected by the absence of indexes on the base tables. If indexes are present, then a different choice of algorithms for joins may help speed up processing, but if indexes are not present the approach can still be used and provide good performance (one caveat is that, to use hash joins, the correlated predicates must contain at least one equality). We noticed that for one level queries, if indexes exist, sometimes performing nested iteration over the subquery is more efficient than using the nested relational approach. The reason is that nested iteration does not need to access base tables, while the nested relational approach has to. However, for multi-level queries, even though indexes exist, performing nested iteration is very time consuming; in our experiments, it is always worse than the nested relational approach.

In our experiments, we implement the nested relational approach on top of a commercial DBMS. It is really necessary to address issues of implementation inside RDBMS. It may be argued that it would be extremely complex to implement this approach in a traditional RDBMS engine, since it would demand extensive rework of storage and algorithms. However, the nested relational algebra is an extension of the relational algebra, so that much of the existing ideas (and code) can be reused (and our version of nested algebra is purposefully restricted). Further, the only new operators are nest and linking selection: nest is implementable with group-by algorithms (either through hashing or sorting), and linking selection can be implemented by scanning (with a slight modification of existing algorithms) or using some of the set-based algorithms (e.g., Mamoulis [2003]). Therefore, the nested relational approach does not require specialized, tailored operators, or "significant" deviations from the relational

framework. There is another issue of storing subrelations, which is far from trivial. Basically, a subrelation can be stored either inline with other attributes or in a separate storage table (the latter seems to be Oracle's approach). In our experiments, because the nested relations are created over intermediate results and the nested relational approach was to create a layer on top of the RDBMS, we chose inline storage – with sorting implementing the nesting. It would be highly interesting to compare how this approach fares with an approach that uses separate storage, and we leave it for further research.

8. CONCLUSION

In this article, we propose the nested relational approach to evaluate nested queries containing nonaggregate subqueries. Unlike traditional approaches, the nested relational approach is based on the nested relational algebra. Our version of the nested relational algebra is a powerful as well as simple language for SQL query processing. We also investigate algebraic optimizations of the operators used in the nested relational approach. Our experiments have shown that the nested relational approach is an efficient and uniform approach that treats all nested queries with any type of linking operators and any level of nesting in a uniform manner. Most importantly, the nested relational approach can be implemented either on top of RDBMS using stored procedures or inside RDBMS by modifying existing algorithms.

ELECTRONIC APPENDIX

The Electronic Appendix for this article can be accessed in the ACM Digital Library. The appendix contains proofs of the lemmas proposed in Section 5 and Section 6, and detailed experiment performance analysis in Section 7.

REFERENCES

ABITEBOUL, S. AND BIDOIT, N. 1984. Non first normal form relations to represent hierarchically organized data. In *Proceedings of the PODS Conference*. 191–200.

AKINDE, M. O. AND BOHLEN, M. H. 2001. Generalized MD-joins: Evaluation and reduction to SQL. In *Proceedings of the VLDB International Workshop on Databases in Telecommunications*. 52–67.

AKINDE, M. AND BOHLEN, M. 2003. Efficient computation of subqueries in complex OLAP. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamiton CA, 163–174.

BADIA, A. 2003a. Automatic generation of XML from relations: The nested relation approach. In *Proceedings of the ER Workshop*. 330–341.

BADIA, A. 2003b. Computing SQL subqueries with Boolean aggregates. In *Proceedings of the DAWAK Conference*. 391–400.

BAEKGAARD, L. AND MARK, L. 1995. Incremental computation of nested relational query expressions. *ACM Trans. Datab. Syst. 20*, 2, 111–148.

BULTINGSLOEWEN, G. V. 1987. Translating and optimizing SQL queries having aggregates. In *Proceedings of the Conference on Very Large Data Bases*. 235–243.

CERI, S. AND GOTTLOB, G. 1985. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Trans. Softw. Eng. 11*, 4, 324–345.

CHATZIANTONIOU, D., AKINDE, M. O., JOHNSON, T., AND KIM, S. 2001. MD-join: an operator for complex OLAP. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamiton, CA, 524–533.

CODD, E. F. 1970. A relational model of data for large shared data banks. *Commun. ACM 13*, 6, 377–387.

COLBY, L. S. 1989. A recursive algebra and query optimization for nested relations. In *Proceedings of the SIGMOD Conference*. ACM, New York, 273–283.

DAYAL, U. 1987. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the Conference on Very Large Data Bases*. 197–208.

GALINDO-LEGARIA, C. A. AND JOSHI, M. M. 2001. Orthogonal optimization of subqueries and aggregation. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 571–581.

GALINDO-LEGARIA, C. AND ROSENTHAL, A. 1997. Outerjoin simplification and reordering for query optimization. *ACM Trans. Datab. Syst. 22*, 1, 43–74.

GANSKI, R. A. AND WONG, H. K. T. 1987. Optimization of nested SQL queries revisited. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 23–33.

GARANI, G. AND JOHNSON, R. 2000. Joining nested relations and subrelations. *Inf. Syst. 25*, 4, 287–307.

GUPTA, A., HARINARAYAN, V., AND QUASS, D. 1995. Aggregate-query processing in data warehousing environments. In *Proceedings of the Conference on Very Large Data Bases*. 358–369.

GYSSENS, M. AND VAN GUCHT, D. 1988. The powerset algebra as a result of adding programming constructs to the nested relational algebra. In *Proceedings of the SIGMOD Conference*. ACM, New York, 225–232.

GYSSENS, M. AND VAN GUCHT, D. 1989. A uniform approach toward handling atomic and structured information in the nested relational database model. *J. ACM 36*, 4, 790–825.

HELMER, S. AND MOERKOTTE, G. 1997. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of the Conference on Very Large Data Bases*. 386–395.

JAESCHKE, G. AND SCHEK, H. J. 1982. Remarks on the algebra of non first normal form relationsl. In *Proceedings of the PODS Conference*. ACM, New York, 124–138.

JAN, Y. 1990. Algebraic optimization for nested relations. In *Proceedings of the ICSS Conference*. 278–287.

KIM, W. 1982. On optimizing an SQL-like nested query. *ACM Trans. Datab. Syst. 7*, 3, 443–469.

LEE, D., MANI, M., CHIU, F., AND CHU, W. W. 2001. Nesting-based relational-to-XML schema translation. In *Proceedings of the WebDB Workshop*. 61–66.

LEVENE, M. AND LOIZOU, G. 1993. Semantics for null extended nested relations. *ACM Trans. Datab. Syst. 18*, 3, 414–459.

LEVENE, M. AND LOIZOU, G. 1994. The nested universal relation data model. *J. Comput. Syst. Sci. 49*, 3, 683–717.

LIU, H.-C. AND RAMAMOHANARAO, K. 1994. Algebraic equivalences among nested relational expressions. In *Proceedings of the CIKM Conference*. 234–243.

LIU, H.-C. AND YU, J. X. 2005. Algebraic equivalences of nested relational operators. *Inf. Syst. 30*, 167–204.

MAKINOUCHI, A. 1977. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of the Conference on Very Large Data Bases*. 447–453.

MAMOULIS, N. 2003. Efficient processing of joins on set-valued attributes. In *Proceedings of the SIGMOD Conference*. ACM, New York, 157–168.

MELNIK, S. AND GARCIA-MOLINA, H. 2002. Divide-and-conquer algorithm for computing set containment joins. In *Proceedings of the EDBT Conference*. 427–444.

MELNIK, S. AND GARCIA-MOLINA, H. 2003. Adaptive algorithms for set containment joins. *ACM Trans. Datab. Sys. 28*, 1, 56–99.

MUMICK, I. S., FINKEKSTEIN, S. J., PIRAHESH, H., AND RAMAKRISHNAN, R. 1990. Magic is relevant. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 247–258.

MUMICK, I. S. AND PIRAHESH, H. 1994. Implementation of magic-sets in a relational database system. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 103–114.

MURALIKRISHNA, M. 1989. Optimization and dataflow algorithms for nested tree queries. In *Proceedings of the Conference on Very Large Data Bases*. 77–85.

MURALIKRISHNA, M. 1992. Improved unnesting algorithms for join aggregate SQL queries. In *Proceedings of the Conference on Very Large Data Bases*. 91–102.

OZSOYOGLU, G., OZSOYOGLU, Z. M., AND MATOS, V. 1987. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. Datab. Syst. 12*, 4, 566–592.

PAREDAENS, J., DE BRA, P., GYSSENS, M., AND VAN GUCHT, D. 1989. *The Structure of the Relational Model*. Springer-Verlag, New York.

RAMASAMY, K., PATEL, J. M., NAUGHTON, J. F., AND KAUSHIK, R. 2000. Set containment joins: The good, the bad and the ugly. In *Proceedings of the SIGMOD Conference*. ACM, New York, 351–362.

RAO, J., LINDSAY, B., LOHMAN, G., PIRAHESH, H., AND SIMMEN, D. 2001. Using EELs, a practical approach to outerjoin and antijoin reordering. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamiton, CA, 585–594.

RAO, J. AND ROSS, K. A. 1998. Reusing invariants: a new strategy for correlated queries. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 37–48.

ROTH, M. A., KORTH, H. F., AND SILBERSCHATZ, A. 1988. Extended relational algebra and calculus for nested relational databases. *ACM Trans. Datab. Syst. 13*, 4, 389–417.

ROTH, M. A., KORTH, H. F., AND SILBERSCHATZ, A. 1989. Null values in nested relational databases. *Acta Inf. 26*, 7, 615–642.

ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBE, S. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the SIGMOD Conference*. ACM, New York, 249–260.

SCHEK, H. J. AND SCHOLL, M. H. 1986. The relational model with relation-valued attributes. *Inf. Syst. 11*, 2, 137–147.

SCHOLL, M. H. 1986. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *Proceedings of the ICDT Conference*. 380–396.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 23–34.

SESHADRI, P., HELLERSTEIN, J. M., PIRAHESH, H., LEUNG, T. Y. C., RAMAKRISHNAN, R., SRIVASTAVA, R., STUCKEY, P. J., AND SUDARSHAN, S. 1996a. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 435–446.

SESHADRI, P., PIRAHESH, H., AND LEUNG, T. Y. C. 1996b. Complex query decorrelation. In *Proceedings of the ICDE Conference*. 450–458.

THOMAS, S. J. AND FISCHER, P. C. 1986. Nested relational structures. *Adv. Comput. Res. 3*, 269–307.

TRANSACTION PROCESSING PERFORMANCE COUNCIL. The TPC-H benchmark. http://www.tpc.org/tpch.

VAN GUCHT, D. 1987. On the expressive power of the extended relational algebra for the unnormalized relational model. In *Proceedings of the PODS Conference*. 302–312.

VAN GUCHT, D., AND FISCHER, P. C. 1986. Some classes of multilevel relational structures. In *Proceedings of the PODS Conference*. 60–69.

VOSSEN, G. 1991. *Data Models, Database Language and Database Management Systems*. Addison-Wesley, Reading, MA.

YAN, W. P., AND LARSON, P. 1994. Performing group-by before join. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamiton, CA, 89–100.

ZHANG, C., NAUGHTON, J., DEWITT, D., LUO, Q., AND LOHMAN, G. 2001. On supporting containment queries in relational database management systems. In *Proceedings of the Conference on Very Large Data Bases*. 425–436.

ZUZARTE, C., PIRAHESH, H., MA, W., CHENG, Q., LIU, L., AND WONG, K. 2003. WinMagic: Subquery elimination using window aggregation. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 652–656.

Received September 2006; revised April 2007; accepted May 2007