On Optimizing an SQL-like Nested Query

WON KIM
IBM Research

SQL is a high-level nonprocedural data language which has received wide recognition in relational databases. One of the most interesting features of SQL is the nesting of query blocks to an arbitrary depth. An SQL-like query nested to an arbitrary depth is shown to be composed of five basic types of nesting. Four of them have not been well understood and more work needs to be done to improve their execution efficiency. Algorithms are developed that transform queries involving these basic types of nesting into semantically equivalent queries that are amenable to efficient processing by existing query-processing subsystems. These algorithms are then combined into a coherent strategy for processing a general nested query of arbitrary complexity.

Categories and Subject Descriptors: H.2.4 [**Information Systems**]: Systems—*query processing*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Relational database, nested query, join, divide, aggregate function, predicate

1. INTRODUCTION

A principal advantage of the relational model of data [7-10] is that it allows the user to express the desired results of a query in a high-level nonprocedural data language without specifying the access paths to stored data. E.F. Codd proposed the relational calculus [7] and the relational algebra [10] for concisely specifying a complex query against the database. However, the mathematics of the relational calculus and the relational algebra is not easy for nontechnical users of a database system to grasp, and their use as data languages may be limited. As a result, much work has been done to develop a data language which is as powerful as the relational calculus and the relational algebra and which is easy for nontechnical users to learn and use. One such data language that has come to be well received is SQL [5, 6]. SQL is a block-structured, calculus-based language which has been implemented in the SEQUEL system [1], System R [2], and ORACLE [23]. SQL-like data languages have also been implemented in ZETA [11, 15], MRDS [22], and DB-85 [14].

Much of this work was done as a part of this author's Ph.D. thesis research in the Computer Science Department at the Universi y of Illinois, Urbana-Champaign, and was supported in part by an IBM graduate fellowship and in part by National Science Foundation Grant US NSF-MCS80-01561.
Author's address: IBM Research, 5600 Cottle Road, San Jose, CA 95193.

One of the more interesting features of SQL is the nesting of query blocks to an arbitrary depth. Without this capability, the power of SQL is severely restricted. However, techniques which have been used to implement this feature in existing systems are in general inefficient and, in view of the popularity of SQL-like data languages, it is imperative to develop efficient methods of processing nested queries. This paper shows that a query nested to an arbitrary depth is composed of five basic types of nesting and that the reason for the less-than-satisfactory performance of nested queries in existing systems is that four of the five basic types of nesting have not been well understood. It develops algorithms which transform nested queries to equivalent, nonnested queries which the query-processing subsystems (often called the *optimizers*) are designed to process more efficiently. The algorithms are based on alternative ways of interpreting the operations of queries which involve the four types of nesting, and may often improve the performance of nested queries by orders of magnitude. Further, these algorithms are directly applicable to the processing of queries expressed in such highly developed, non-SQL-like data languages as QUEL, which is supported by INGRES [21].

This paper presumes that the reader is familiar with the syntax of SQL, as well as with the basic concepts and terminology of relational databases.

The illustrative examples used in this paper are based on the following database of suppliers, parts, projects, and shipments:

SUPPLIER (SNO, SNAME, SLOC, SBUDGET)
PART (PNO, PNAME, DIM, PRICE, COLOR)
PROJECT (JNO, JNAME, PNO, JBUDGET, JLOC)
SHIPMENT (SNO, PNO, JNO, QTY)

Each SUPPLIER tuple contains the number (identifier), name, location, and budget of a supplier. Each PART tuple contains the number (identifier), name, dimension, unit price, and color of a part. A PROJECT tuple has fields for the number (identifier), name, budget, and location of a project, along with the part number of a part it uses. A SHIPMENT tuple has fields for a supplier number, along with the part number, project number, and quantity of the part the supplier supplies.

Throughout this paper R_i denotes a relation in the database, C_k the kth column of a relation, and B the size in pages of available main-memory buffer space.

This paper is organized as follows: Section 2 presents a new scheme for classifying a nested query into one of five basic types. Algorithms for transforming a nested query of one of four basic nesting types into its semantically equivalent, nonnested form are developed and illustrated in Sections 3–5. Section 6 indicates potential performance improvements that these algorithms can bring about. Section 7 shows how these algorithms may be merged into a coherent strategy for processing a general SQL-like nested query of arbitrary complexity.

2. NESTING OF QUERY BLOCKS

The fundamental structure of an SQL-like query is syntactically represented by a *query block* which consists of SELECT, FROM, and WHERE clauses. The WHERE clause specifies the predicates which tuples of the relation indicated in

the FROM clause must satisfy. A predicate is of the form [$R_i.C_k$ **op** X], where X is a constant or a list of constants, and **op** is a scalar comparison operator (=, ¬=, >, >=, <, <=) or a set membership operator (IS IN, IS NOT IN). The SELECT clause specifies the columns of the relation to be output and operations on the columns. The following two examples illustrate the basic structure of an SQL-like query.

Example 1. Find the supplier numbers of suppliers which supply parts whose part number is P1.

```
SELECT   SNO
FROM     SHIPMENT
WHERE    PNO = 'P1';
```

Example 2. Find the highest part number of parts whose unit price is greater than 25.

```
SELECT   MAX(PNO)
FROM     PART
WHERE    PRICE > '25';
```

As long as the predicates in the WHERE clause are restricted to the simple predicates of the form [$R_i.C_k$ **op** X], only single-relation queries can be formulated. If n-relation queries which require relational join and division operations are to be formulated, more general predicates must be allowed. The simple predicates can only be extended in three ways.

(1) *Nested predicate.* X may be replaced by Q, an SQL-like query block, to yield a predicate of the form [$R_i.C_k$ **op** Q]. The **op** may be a scalar or set membership operator. The predicate may also take the form [Q **op** $R_i.C_k$]. The set membership operator is then replaced by the set containment operator (CONTAINS, DOES NOT CONTAIN). The two forms of the predicate are symmetric, so the latter will not be further considered.

(2) *Join predicate.* X may be replaced by $R_j.C_h$ to yield a predicate of the form [$R_i.C_k$ **op** $R_j.C_h$], in which R_i ¬= R_j and the **op** is a scalar comparison operator.

(3) *Division predicate.* $R_i.C_k$ and X may be replaced by query blocks Q_i and Q_j, respectively, to yield a predicate of the form [Q_i **op** Q_j]. The **op** may be a scalar comparison operator, set comparison operator (=, ≠), or set membership and containment operator. As will be seen later, Q_j (or Q_i) may be a constant or a list of constants, provided that Q_i (or Q_j) has in its WHERE clause a join predicate which references the relation of the outer query block.

For simplicity, the **op** in a join predicate is assumed to be an equality operator in this paper. The join predicate [$R_i.C_k = R_j.C_h$] indicates that each tuple of R_i is to be joined with all tuples of R_j whose C_h column values are equal to the C_k column value of the R_i tuple. The next example illustrates the use of a join predicate in formulating a two-relation query.

Example 3. Find the supplier numbers of suppliers which supply parts whose unit price is higher than 25:

```
SELECT   SNO
FROM     SHIPMENT, PART
WHERE    SHIPMENT.PNO = PART.PNO   AND
         PRICE > '25';
```

Note that a query which requires joining two relations has been expressed using one join predicate. A query which joins n relations can be written using a conjunction of $n - 1$ join predicates in the WHERE clause. An n-relation query which joins n relations via $n - 1$ join predicates and contains no nested or division predicates is termed a *canonical n-relation query.* As shown later in this section, it is possible to write a query which joins relations without the use of join predicates.

Examples 4 through 7 later in this section illustrate uses of nested predicates, and Example 8 shows a query which makes use of a division predicate.

Besides providing the capability for expressing a canonical n-relation query, the generalized predicates allow the user to nest query blocks to an arbitrary depth. For the following discussion, however, a query is assumed to be of nesting depth one. That is, a query consists of an outer block and an inner block. (The query blocks Q_i and Q_j in a division predicate are the left- and right-hand sides of the inner block.) Further, it is assumed that the WHERE clause of the outer block contains only one nested predicate or division predicate. These assumptions cause no loss of generality, as is shown in Section 7. Finally, as in SQL, it is assumed that only one column name may be specified in the SELECT clause of a query block that is nested within another query block.

It is clear that, of the three types of generalized predicates defined above, only the nested predicate and the division predicate give rise to the nesting of query blocks. A nested predicate may cause one of four basic types of nesting, according to whether the inner query block Q has in the WHERE clause a join predicate that references the relation of the outer query block and whether the column name in the SELECT clause of Q has associated with it an aggregate function (SUM, AVG, MAX, MIN, COUNT). A division predicate yields a fifth basic nesting. It appears that the present specification of SQL gives rise to no other basic nesting types. However, it does not appear that an extra nesting type will enhance the power of SQL-like data languages. The remainder of this section will illustrate these five basic nesting types and, for later use, develop graphical representations for them.

A nested predicate gives rise to *type-A nesting* if the inner query block Q does not contain a join predicate that references the relation of the outer query block and the SELECT clause of Q indicates an aggregate function associated with the column name. Note that if Q does not contain in the WHERE clause a join predicate that references the relation of the outer query block, Q may be completely evaluated independently of the outer query block. Moreover, when the SELECT clause of Q specifies an aggregate function, the result of evaluating

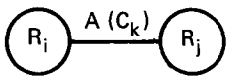
Q will be a single constant rather than a list of constants. A type-A nesting of depth 1 is illustrated by the following example.

Example 4. Find the supplier numbers of suppliers which supply parts whose part number is the highest of those parts whose unit price is greater than 25.

```
SELECT    SNO
FROM      SHIPMENT
WHERE     PNO = (SELECT    MAX(PNO)
                 FROM      PART
                 WHERE     PRICE > '25');
```

There is only one way to process a type-A nested query on a single processor. The inner block has to be evaluated first. The nested predicate of the outer block then becomes a simple predicate, since Q can be replaced by a constant. The outer block then is no longer nested and can be processed completely.

A type-A nested query (predicate) of depth 1 is graphically represented as follows. The graphical representation for a query will be called the *query graph* for the query.



The left-hand node represents the outer block (which requires access to R_i), and the right-hand node the inner block (which references R_j). The arc represents the nested predicate and is labeled 'A' followed by the column name that appears in the SELECT clause of the inner query block. In general, if the WHERE clause of the outer block consists of n nested predicates, n arcs leading to n right-hand nodes emanate from one left-hand node.

A nested predicate yields a *type-N nesting*, if Q does not contain a join predicate that references the relation of the outer query block and the column name in the SELECT clause of Q does not have an aggregate function associated with it. The result of evaluating the Q of a type-N nested predicate is a list of constants, and, as in a type-A nested query, Q may be completely evaluated independently of the outer query block. The following example illustrates a type-N nesting.

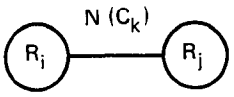
Example 5. Find the supplier numbers of suppliers which supply parts whose unit price is greater than 25.

```
SELECT    SNO
FROM      SHIPMENT
WHERE     PNO  IS  IN  (SELECT    PNO
                        FROM      PART
                        WHERE     PRICE > '25');
```

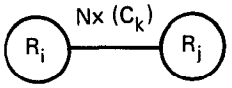
Note that Examples 3 and 5 show two different formulations of the same query. This observation has some interesting implications, as can be seen in Section 3.

The System R approach to processing the above query is first to process the inner query block Q in order to replace the type-N nested predicate, PNO IS IN Q, with a simple predicate PNO IS IN X, where X is the result of evaluating Q, and then to completely process the resulting nonnested query.

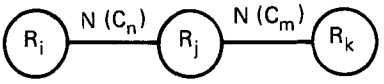
The query graph for a type-N nested query of depth 1 is



A type-N nested query in which the **op** of the nested predicate is the set noninclusion operator (IS NOT IN) requires a different treatment from a type-N nested query involving the set inclusion operator (IS IN), as is shown in Section 3. This special type-N nested query is represented by a straight arc labeled 'N*x*'.



A type-N nested query of depth $n (n >= 1)$ is defined as a nested query whose query graph consists of only straight arcs labeled 'N'. The following is the query graph for a type-N nested query of depth 2.

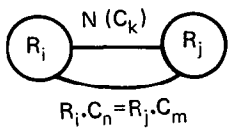


A nested predicate gives rise to a *type-J nesting* when the WHERE clause of Q contains a join predicate that references the relation of the outer query block and the column name in the SELECT clause of Q is not associated with an aggregate function. The next example illustrates a type-J nesting.

Example 6. Find the supplier numbers of suppliers which supply projects that are located in New York and whose project numbers are the same as the supplier numbers.

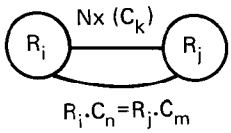
```
SELECT   SNO
FROM     SHIPMENT
WHERE    PNO  IS IN  (SELECT   PNO
                      FROM     PROJECT
                      WHERE    SHIPMENT.SNO = PROJECT.JNO   AND
                               JLOC = 'NEW YORK');
```

The following is the query graph for a type-J nested query of depth 1.

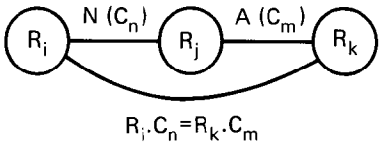


The straight arc is labeled 'N', since except for the presence of a join predicate in the inner block, a type-J nested query has the same form as a type-N nested query. The circular arc, labeled with a join predicate, links the two query blocks that the join predicate references.

The query graph for a type-J nested query of depth 1 involving the set noninclusion operator is shown in the following.



A type-J nested query of depth $n(n >= 1)$ is defined as a nested query whose query graph consists of at least one circular arc and no straight arc labeled 'A'. The following is the query graph for a type-J nested predicate of depth 2.

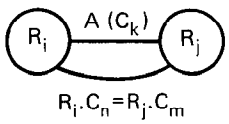


A *type-JA nesting* results from a nested predicate if the WHERE clause of Q contains a join predicate that references the relation of the outer query block and an aggregate function is associated with the column name in the SELECT clause of Q. The next example illustrates a type-JA nesting.

Example 7. Find the supplier numbers of suppliers which supply parts whose part numbers are the highest of those parts used by each of the projects located in New York.

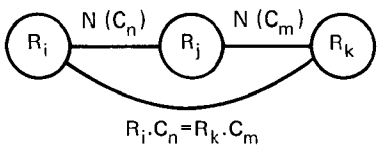
```
SELECT   SNO
FROM     SHIPMENT
WHERE    PNO = (SELECT    MAX(PNO)
                FROM      PROJECT
                WHERE     PROJECT.JNO = SHIPMENT.JNO   AND
                          JLOC = 'NEW YORK');
```

The query graph for a type-JA nested predicate of depth 1 is



The straight arc joining the two nodes is labeled 'A'. The circular arc connects the two query blocks referenced in the join predicate.

A type-JA nested query of depth $n(n >= 1)$ is defined as a nested query whose query graph exhibits at least one circular arc and at least one straight arc labeled 'A', as shown below.



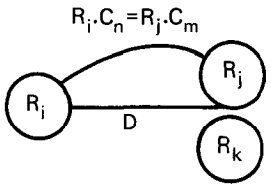
A join predicate and a division predicate together give rise to a *type-D nesting*, if the join predicate in either Q_i or Q_j (or both) references the relation of the outer block. As the next example shows, a type-D nested query expresses the relational division operation.

Example 8. Find the names of suppliers which supply all red parts that are priced higher than 25.

```
SELECT   SNAME
FROM     SUPPLIER
WHERE    (SELECT   PNO
          FROM     SHIPMENT
          WHERE    SHIPMENT.SNO = SUPPLIER.SNO)
                CONTAINS
         (SELECT   PNO
          FROM     PART
          WHERE    COLOR = 'RED'   AND
                   PRICE > '25');
```

The division predicate in this query specifies the division of TMP1 (SNO, PNO) by TMP2 (PNO), where TMP1 is obtained by projecting the SHIPMENT relation over the SNO and PNO columns, and TMP2 by evaluating the right-hand side of the division predicate.

The query graph for a type-D nested query of depth 1 is



The straight arc is labeled 'D'. One straight arc leads to two right-hand nodes, since the inner block consists of two query blocks, Q_i and Q_j. The top right-hand node is the left-hand side of the division predicate. The circular arc connects the outer block and the left-hand side of the division predicate.

Both sides of the inner block of a type-D nested query may contain join predicates which link them to the outer block. However, a division predicate in which neither query block contains a join predicate that references the outer block is meaningless. The reason is that a predicate is a condition which is imposed on tuples of the relation indicated in the FROM clause of the query block, and a division predicate which does not contain a join predicate which references the relation is either always true or always false.

The System R technique for processing a nested query in which the WHERE clause of the inner query block contains a join predicate that references the relation of the outer query block (that is, types J, JA, and D) is the conceptually simple, *nested-iteration method* [16–18, 20, 24]. This method suggests the complete processing of the inner query block for each tuple of the relation of the outer query block. For example, the query of Example 6 may be processed by the nested-iteration method by fetching each SHIPMENT tuple, isolating its SNO column value for substitution in the join predicate of the inner query block, evaluating the inner query block (which by now is free of any join predicate that references the relation of the outer query block), and reducing the type-J nested predicate of the original query to a simple predicate PNO IS IN X, where X is the result of evaluating the inner query block for a particular SNO value of the SHIPMENT tuple.

Although it provides a convenient way of viewing the semantics of a nested query, the nested-iteration method is an inefficient way to evaluate it in a paged-memory environment. In fact, the main thrust of this paper is the development of algorithms for transforming, with some intermediate query processing, a nested query of various types to a semantically equivalent canonical form which can be evaluated more efficiently. This is illustrated in Section 6.

3. PROCESSING A TYPE-N OR TYPE-J NESTED QUERY

Examples 3 and 5 have shown that some query may be expressed in its canonical form or type-N nested form. The equivalence of the two forms of the same query has been recognized by others [see 12, Chap. 7]. However, some important questions which arise have not been investigated. Can a type-N nested query of depth $n - 1$ be transformed to a semantically equivalent canonical n-relation query, and vice versa? Which of the two forms of the query can be more efficiently processed?

First, Lemma 1 establishes the equivalence of the canonical and type-N nested form of a two-relation query in which the **op** is not the set noninclusion operator, IS NOT IN.

Let Q_1 be

```
SELECT   Ri.Ck
FROM     Ri, Rj
WHERE    Ri.Ch = Rj.Cm
```

And let Q_2 be

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Ch  IS IN  (SELECT   Rj.Cm
                        FROM     Rj);
```

LEMMA 1. *Q_1 and Q_2 are equivalent; that is, they yield the same result.*

PROOF. By definition, the inner block of Q_2 can be evaluated independently of the outer block and the result of evaluating it is X, a list of values in the C_m column of R_j. Q_2 is then reduced to

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Ch  IS IN  X;
```

The predicate $R_i.C_h$ IS IN X is satisfied only if X contains a constant x such that $R_i.C_h = x$. That is, it can be satisfied only for those tuples of R_i and R_j which have common values in the C_h and C_m columns, respectively. The join predicate $R_i.C_h = R_j.C_m$ specifies exactly this condition. □

Lemma 1 implies one important assumption. The result of evaluating the inner block of Q_2 is X, a list of values in the C_m column of R_j. Since the list is obtained by projecting R_j over the C_m column, in general it will contain duplicate values. But the effect of the simple predicate $R_i.C_h$ IS IN X is to implicitly remove any redundant values from X. However, the join predicate $R_i.C_h = R_j.C_m$ of Q_1 does not imply removal of duplicate values from the C_m column of R_j and the result of Q_1 reflects their presence. Therefore, it is assumed throughout this paper that, if

a canonical n-relation query has been obtained from a type-N or type-J nested query and *if and only if* the **op** of the nested predicate is IS IN, it is processed by first restricting and projecting the relation which corresponds to the relation of the inner query block in the nested query (e.g., R_j in Q_1) and then removing duplicate values from the resulting unary (i.e., single-column) relation before joining it with the relation which corresponds to the relation of the outer query block in the nested query (e.g., R_i in Q_1). This assumption guarantees the correctness of Lemma 1 and appears to be reasonable, since the relation which results from projecting and restricting a relation is usually much smaller than the initial relation, and the cost of joining reduced relations is, therefore, usually smaller than the cost of joining unreduced relations.

It is also important to recognize that Lemma 1 establishes only that the type-N nested form of a query in which the **op** of the nested predicate is the set inclusion operator can be transformed to its canonical form. Lemma 1 does not apply when the **op** of the nested predicate is the set noninclusion operator, as is shown later. Further, the canonical form of a query cannot in general be transformed to the type-N nested form. This is because only one column (the join column) is allowed in the SELECT clause of the inner block [6]. If, for example, the SELECT clause of Q_1 contained $R_j.C_q$ as well as $R_i.C_k$, it would not be possible to express the query in the nested form. Lemma 1 can be readily extended to type-N nested predicate in which the **op** is a scalar comparison operator.

The scalar comparison operator becomes the **op** of the join predicate in the canonical form of the query. It is assumed that the result of evaluating the inner query block is always a single constant. If it is desirable to be able to abort the query when the inner query block yields a list of constants, the canonical equivalent must be processed as if the **op** is IS IN.

Lemma 1 suggests Algorithm NEST-N-J for transforming a type-N nested query of depth $n - 1$ to its canonical form.

Algorithm NEST-N-J

(1) Combine the FROM clauses of all query blocks into one FROM clause.
(2) AND the WHERE clauses of all query blocks into one WHERE clause.
(3) Replace [$R_i.C_h$ **op** (SELECT $R_j.C_m$] by a join predicate [$R_i.C_h$ **new-op** $R_j..C_m$], and AND it to the combined WHERE clause obtained on step 2. Note that if **op** is IS IN, the corresponding **new-op** is '='; otherwise, **new-op** is the same as **op**.
(4) Retain the SELECT clause of the outermost query block.

Algorithm NEST-N-J attaches no significance to join predicates. A type-J nested query can also be transformed to its canonical form by the algorithm. The join predicate in the inner block which references the outer block is ANDed to other simple predicates on step 2. The following example illustrates Algorithm NEST-N-J.

Example 9

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Ch  IS IN  (SELECT   Rj.Cm
                         FROM     Rj
                         WHERE    Ri.Cn = Rj.Cp);
```

The canonical form of this query is shown below:

```
SELECT   Ri.Ck
FROM     Ri, Rj
WHERE    Ri.Ch = Rj.Cm  AND
         Ri.Cn = Rj.Cp;
```

The query joins relations R_i and R_j, on (C_h, C_n) and (C_m, C_p) columns, respectively.

The process of transforming a type-N or type-J nested query of depth n to its canonical equivalent can be viewed as one of eliminating all straight or circular arcs as well as all but the leftmost node on the query graph for the nested query.

A type-N nested query in which the **op** in the nested predicate is the set noninclusion operator requires a careful, special consideration. Consider the following query.

Example 10. Find the names of suppliers that do *not* supply parts whose part number is P1.

```
SELECT   SNAME
FROM     SUPPLIER
WHERE    SNO  IS NOT IN  (SELECT   SNO
                          FROM     SHIPMENT
                          WHERE    PNO = 'P1');
```

It is important to understand that the following query is not equivalent to the above query.

```
SELECT   SNAME
FROM     SUPPLIER
WHERE    SNO  IS IN  (SELECT   SNO
                      FROM     SHIPMENT
                      WHERE    PNO ¬= 'P1');
```

Further, as pointed out in [12], the operation of the query may be viewed as fetching each SUPPLIER tuple, checking whether the supplier supplies part P1, and, if not, outputting the supplier name. This alternate interpretation of the query results in the formulation of a type-D nested query.

```
SELECT   SNAME
FROM     SUPPLIER
WHERE    (SELECT   PNO
          FROM     SHIPMENT
          WHERE    SHIPMENT.SNO = SUPPLIER.SNO)
                   DOES NOT CONTAIN
                   P1;
```

As is shown in Section 5, efficient processing of a type-D nested query in general requires sorting the relation of the left-hand side of the inner block, the SHIPMENT relation in the present example. The equivalent type-N nested form of the query, on the other hand, requires sequentially scanning the SHIPMENT relation only once. Therefore, the type-D nested form of the query is not helpful. However, the type-N nested form also suffers if X, the result of evaluating the inner block, is large compared with B, the size of main-memory buffer space.

A potentially better method of processing the above query when the result of evaluating the inner block is large is a special merge join of X and SUPPLIER. Both relations are first sorted in SNO-column order and then simultaneously scanned to find SUPPLIER tuples which have SNO values that are *not* shared by X. X and SUPPLIER may be said to be '*antijoined*' by the antijoin predicate ¬(SUPPLIER.SNO = X.SNO). The above query may then be transformed to its pseudocanonical form as follows.

```
SELECT   SNAME
FROM     SUPPLIER, X
WHERE    ¬(SUPPLIER.SNO = X.SNO);
```

The set noninclusion operator in a type-N or type-J nested query requires an extension to Algorithm NEST-N-J. Now [$R_i.C_k$ IS NOT IN (SELECT $R_j.C_h$] is replaced by the antijoin predicate $\neg(R_i.C_k = R_j.C_h)$ and ANDed to the merged WHERE clause obtained on steps 2 and 3 of the algorithm.

The result of a query is independent of the order in which its predicates are evaluated. However, an antijoin predicate must be evaluated only after all join predicates have been evaluated. The following query explains this point.

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Ch  IS NOT IN  (SELECT   Rj.Cm
                            FROM     Rj
                            WHERE    Rj.Cn = Ri.Cp);
```

The operation of the query may be thought of as fetching each tuple of R_i, retrieving all tuples of R_j whose C_n column values are the same as the C_p column value of the R_i tuple (i.e., if $R_j.C_n = R_i.C_p$), and outputting the C_k column of the R_i tuple if its C_h column value is not found in the C_m column of the retrieved R_j tuples (i.e., if $\neg(R_i.C_h = R_j.C_m)$). Notice how the join predicate, $R_j.C_n = R_i.C_p$, is evaluated before the antijoin predicate, $\neg(R_i.C_h = R_j.C_m)$ (implied in $R_i.C_h$ IS NOT IN (SELECT $R_j.C_m$) for each value in the C_p column of R_i.

4. PROCESSING A TYPE-JA NESTED QUERY

A new algorithm for processing a type-JA nested query is developed in this section. The algorithm not only improves the performance of a type-JA nested query but also appears to be easy to implement. Lemma 2 provides the basis for the algorithm. An algorithm similar to the one described in this section has been independently developed by Epstein in the context of the QUEL data language [13].

Let Q_3 be

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Ch = (SELECT   AGG(Rj.Cm)
                  FROM     Rj
                  WHERE    Rj.Cn = Ri.Cp);
```

Further, let Q_4 be

SELECT $R_i.C_k$
FROM R_i
WHERE $R_i.C_h$ = (SELECT $R_t.C_2$
FROM R_t
WHERE $R_t.C_1 = R_i.C_p$);

where $R_t(C_1, C_2)$ is obtained by

$R_t(C_1, C_2)$ = (SELECT $R_j.C_n$, AGG($R_j.C_m$)
FROM R_j
GROUP BY $R_j.C_n$);

LEMMA 2. *Q_3 and Q_4 are equivalent, that is, they produce the same result.*

PROOF. It is shown in Section 2 that the operation of Q_3 may be thought of as first fetching a tuple of R_i and all tuples of R_j whose C_n column values are the same as the C_p column value of the R_i tuple, then applying the aggregate function AGG on the C_m column of the R_j tuples to obtain a constant x, and, finally, outputting the C_k value of the R_i tuple if $x = C_h$ column value of the R_i tuple. Now R_t is a binary relation of each distinct value in the C_n column of R_j and the corresponding value obtained by applying the aggregate function AGG on the R_j tuples. Then it is clear that the query may be processed by fetching each tuple of R_i, then fetching the R_t tuple whose C_1 column has the same value as the C_p column of the R_i tuple, and outputting the C_k column value of the R_i tuple if the C_2 column value of the R_t tuple is the same as the C_h value of the R_i tuple. But this is exactly the operation of Q_4. □

Note that Q_3 is type-JA nested, but Q_4 is type-J nested and can be transformed to its canonical form by Algorithm NEST-N-J. Since the **op** of the type-J nested predicate is the scalar equality operator, rather than the set inclusion operator IS IN, the problem of removing duplicates from R_t does not arise here. In other words, the straight arc labeled 'A' on the query graph for Q_3 is replaced by a straight arc labeled 'N', and the circular arc is retained.

Lemma 2 directly leads to an algorithm which transforms a type-JA nested query of depth 1 to an equivalent type-J nested query of depth 1. Consider the following query.

SELECT $R_1.C_{n+2}$
FROM R_1
WHERE $R_1.C_{n+1}$ = (SELECT AGG($R_2.C_{n+1}$)
FROM R_2
WHERE $R_2.C_1 = R_1.C_1$ AND
$R_2.C_2 = R_1.C_2$ AND
⋮
$R_2.C_n = R_1.C_n$);

Algorithm NEST-JA

(1) Generate a temporary relation $R_t(C_1, \ldots, C_n, C_{n+1})$ from R_2 such that each C_{n+1} column value of R_t is a constant obtained by applying the aggregate function AGG on

the C_{n+1} column of the R_2 tuples which share a common value in columns C_1 through C_n. In other words, the primary key of R_t is its first n columns. R_t can be obtained by

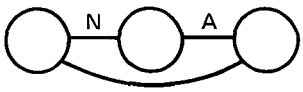
$$R_t(C_1, \ldots, C_n, C_{n+1}) = (\text{SELECT } C_1, \ldots, C_n, \text{AGG}(C_{n+1}) \text{ FROM } R_2 \text{ GROUP BY } C_1, \ldots, C_n);$$

(2) Transform the inner block of the initial query by changing all references to R_2 columns in join predicates that reference R_1 to corresponding R_t columns. The initial type-JA nested query is now type-J nested, since the aggregate function in the SELECT clause has been replaced by a simple reference to the C_{n+1} column of R_t.

```
SELECT   R1.Cn+2
FROM     R1
WHERE    R1.Cn+1 = (SELECT   Rt.Cn+1
                    FROM     Rt
                    WHERE    Rt.C1 = R1.C1   AND
                             Rt.C2 = R1.C2   AND
                               .
                               .
                               .
                             Rt.Cn = R1.Cn);
```

In order to extend Lemma 2 to an algorithm for transforming a type-JA nested query of depth $n (n >= 1)$ to an equivalent, type-J nested query, it is instructive to consider first the transformation of a type-JA nested query of depth 2. A type-JA nested query of depth 2 has at least one circular arc on its query graph and one or both of the straight arcs must be labeled 'A'. Therefore, there are 3 types of type-JA nested query of depth 2, as illustrated below.

type JA(NA)



type JA(AA)

type JA(AN)

It is clear that Algorithm NEST-JA, when applied to the two right-hand nodes, transforms a type-JA(NA) nested query to an equivalent type-J query of depth 2 shown below.

Similarly, it transforms a type-JA(AA) nested query to an equivalent type-JA(AN) nested query. A type-JA(AN) nested query can be transformed to a type-JA query of depth 1 if Algorithm NEST-N-J is first applied to the two innermost blocks (i.e., the two rightmost nodes on the query graph). The next example illustrates this.

Example 11. The following is a type-JA(AA) nested query of depth 2.

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Ch = (SELECT    AGG(Rj.Cm)
                  FROM      Rj
                  WHERE     Rj.Cn =    (SELECT    AGG(Rk.Cp)
                                        FROM      Rk
                                        WHERE     Rk.Cq = Ri.Cr));
```

First, a temporary relation $R_{t1}(C_1, C_2)$ is obtained by step 1 of Algorithm NEST-JA.

```
Rt1(C1, C2) = (SELECT     Rk.Cq, AGG(Rk.Cp)
               FROM       Rk
               GROUP BY   Rk.Cq);
```

By step 2 of Algorithm NEST-JA, the innermost block is transformed to

```
SELECT   Rt1.C2
FROM     Rt1
WHERE    Rt1.C1 = Ri.Cr;
```

The initial type-JA(AA) nested query is now type-JA(AN) nested. The two inner blocks of the type-JA(AA) nested query can then be transformed to its canonical form by Algorithm NEST-N-J, yielding the following type-JA nested query of depth 1.

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Ch = (SELECT    AGG(Rj.Cm)
                  FROM      Rj, Rt1
                  WHERE     Rt1.C1 = Ri.Cr   AND
                            Rt1.C2 = Rj.Cn);
```

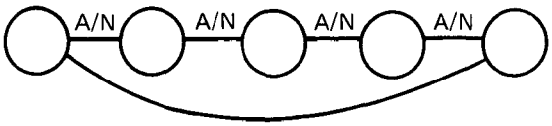
Next, another temporary relation, $R_{t2}(C_1, C_2)$, is obtained by step 1 of Algorithm NEST-JA.

```
Rt2(C1, C2) = (SELECT     Rt1.C1, AGG(Rj.Cm)
               FROM       Rt1, Rj
               WHERE      Rj.Cn = Rt1.C2
               GROUP BY   Rt1.C1);
```

By another application of step 2 of Algorithm NEST-JA, the initial query is transformed to an equivalent type-J query of depth 1.

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Ch = (SELECT   Rt2.C2
                  FROM     Rt2
                  WHERE    Rt2.C1 = Ri.Cr);
```

An algorithm for transforming a type-JA query of depth n to an equivalent type-J nested query can now be given. The following query graph for a type-JA nested query provides the framework for the description of Algorithm NEST-JA(G).

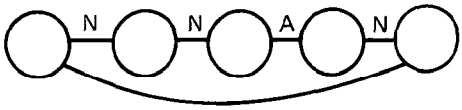


(The label "A/N" over each straight arc means A or N. Remember at least one straight arc must be labeled "A" if the query is type-JA nested.)

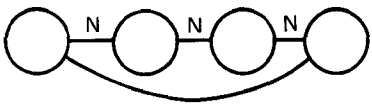
Algorithm NEST-JA(G)

```
I = n (the nesting depth of the query);
DO WHILE (there-is-at-least-one-straight-arc-labeled-A);
If the Ith straight arc is labeled N THEN I = I - 1;
ELSE DO;
      Apply Algorithms NEST-JA and (NEST-N-J) to the n - I + 1 nodes to the right
          of the Ith straight arc;
      The Ith straight arc of the resulting query of depth I is labeled N;
      n = I;
      END;
END;
```

Example 12. The following type-JA nested query of depth 4 is transformed to an equivalent type-J nested query by Algorithm NEST-JA(G).



First, the fourth straight arc is labeled "N", so set $I = n - 1 = 3$. The third straight arc is labeled "A", so Algorithms NEST-N-J and NEST-JA are applied to the $n - I + 1 = 2$ right-hand nodes of the third straight arc to yield a query of depth 3 shown below.



The query is now type-J nested, and the algorithm terminates.

5. PROCESSING A TYPE-D NESTED QUERY

As far as this author is aware, none of the currently operational relational database systems supports the division predicate. However, it is often necessary to formulate a query that requires division of one relation by another relation, and it does appear that the relational division operation can be efficiently performed. A more efficient algorithm than the nested-iteration method for processing a type-D nested query can be found by, once again, recognizing an alternate interpretation of the operation of the query. The algorithm is based on Lemma 3. Let Q_5 be the following query.

```
SELECT   Ri.Ck
FROM     Ri
WHERE    (SELECT   Rj.Ch
          FROM     Rj
          WHERE    Rj.Cn = Ri.Cp)
                   op
         (SELECT   Rk.Cm
          FROM     Rk);
```

Further, let Q_6 be

```
SELECT   Ri.Ck
FROM     Ri
WHERE    Ri.Cp = (SELECT   C1
                  FROM     Rt);
```

where R_t is obtained as

```
Rt(C1) = (SELECT   Rj.Cn
          FROM     Rj RX
          WHERE    (SELECT   Rj.Ch
                    FROM     Rj RY
                    WHERE    RY.Cn = RX.Cn)
                             op
                   (SELECT   Rk.Cm
                    FROM     Rk);
```

LEMMA 3. *Q_5 and Q_6 are equivalent, that is, they produce the same result.*

PROOF. As has been shown, the operation of Q_5 may be thought of as fetching each tuple of R_i and checking whether the division predicate is satisfied by the C_p column value of the tuple. But what if there is a list of the C_n column values of R_j which satisfy the division predicate? Then all that needs to be done is to fetch each R_i tuple and determine whether the C_p column value of the tuple is in the list. But this is precisely the operation of Q_6, since R_t is just such a list. □

Note that the initial type-D nested query has been transformed to an equivalent type-N nested query. The fact that the **op** of the type-N nested predicate is the scalar equality operator, rather than the set inclusion operator IS IN, means that the type-N nested query may be transformed to its canonical equivalent form without having to remove duplicates from R_t. In terms of query graphs, the straight arc labeled "D" on the query graph for the initial query is replaced by a straight arc labeled "N", the two right-hand-side nodes are replaced by one node,

and the circular arc on the initial query graph is eliminated. It should be clear that the initial query could have been directly transformed to its canonical form. The following example illustrates Lemma 3.

Example 13. Find the names of suppliers which supply all red parts.

```
SELECT   SNAME
FROM     SUPPLIER
WHERE    (SELECT    PNO
          FROM      SHIPMENT
          WHERE     SHIPMENT.SNO = SUPPLIER.SNO)
                =
         (SELECT    PNO
          FROM      PART
          WHERE     COLOR = 'RED');
```

The query is equivalent to

```
SELECT   SNAME
FROM     SUPPLIER
WHERE    SNO  IS IN  (SELECT    SNO
                      FROM      TMP);
```

where the temporary relation TMP(SNO) is obtained by

```
TMP(SNO) = (SELECT    SNO
            FROM      SHIPMENT      SX
            WHERE     (SELECT       PNO
                       FROM         SHIPMENT SY
                                =
                       WHERE        SX.SNO = SY.SNO)
                      (SELECT       PNO
                       FROM         PART
                       WHERE        COLOR = 'RED');
```

The query which generates the temporary relation R_t is simply an SQL-like formulation of the relational division operator. Lemma 3 provides the basis for Algorithm NEST-D, which transforms a general type-D nested query to an equivalent canonical two-relation query. Consider the following query which requires dividing $R_2(C_1, \ldots, C_n)$ by $R_3(C_1, \ldots, C_m)$, where $n > m$.

```
SELECT   R1.C1
FROM     R1
WHERE    (SELECT    R2.C1
          FROM      R2
          WHERE     R2.C2 = R1.C2  AND
                    R2.C3 = R1.C3  AND
                          :
                    R2.Cn = R1.Cn)
                          =
         (SELECT    R3.C1
          FROM      R3
          WHERE     R3.C2 = R1.C2  AND
                    R3.C3 = R1.C3  AND
                          :
                    R3.Cm = R1.Cm);
```

Algorithm NEST-D

(1) Generate a temporary relation $R_{t1}(C_1, \ldots, C_m)$ by

$$R_{t1}(C_1, \ldots, C_m) = (\text{SELECT} \quad C_1, \ldots, C_m \quad \text{FROM} \quad R_3);$$

Also generate a temporary relation $R_{t2}(C_1, \ldots, C_n)$ by

$$R_{t2}(C_1, \ldots, C_n) = (\text{SELECT} \quad C_1, \ldots, C_n \quad \text{FROM} \quad R_2);$$

(2) Divide $R_{t2}(C_1, \ldots, C_n)$ by $R_{t1}(C_1, \ldots, C_m)$. The result is a new temporary relation $R_{t3}(C_{m+1}, \ldots, C_n)$.

(3) Transform the initial query to its canonical equivalent by first dropping the query block on R_3, replacing all references to columns of R_2 to corresponding columns of R_{t3} in the query block on R_2, and eliminating the SELECT and FROM clauses of the query block on R_2. The FROM clause of the resulting canonical query must now include R_{t3} as well.

$$\begin{array}{lll} \text{SELECT} & R_1.C_1 & \\ \text{FROM} & R_1, R_{t3} & \\ \text{WHERE} & R_1.C_{m+1} = R_{t3}.C_{m+1} & \text{AND} \\ & R_1.C_{m+2} = R_{t3}.C_{m+2} & \text{AND} \\ & \vdots & \\ & R_1.C_n = R_{t3}.C_n; & \end{array}$$

6. RATIONALE FOR TRANSFORMING NESTED QUERIES

What is the reason for transforming a nested query to its canonical equivalent? The answer is that the optimizers in currently operational relational database systems that support SQL-like query languages have been designed to evaluate the canonical form of multiple-relation queries and they resort solely to the nested-iteration method for evaluating the nested form of the queries. The nested-iteration method is efficient only for a limited set of query and database characteristics, as is shown in this section.

A generally more effective strategy for evaluating a nested query (of arbitrary depth and complexity) is to transform it to its canonical form and have the optimizer determine an optimal set of algorithms and access paths for evaluating it. The System R optimizer, for example, considers both the nested-iteration method and the *merge-join* method, as well as all possible "reasonable" orders in which relations may be scanned, in processing a canonical n-relation query [19]. Whereas the nested-iteration method of joining two relations requires the inner relation to be retrieved as many times as there are tuples that satisfy predicates on the outer relation, the merge-join requires both relations to be simultaneously retrieved only once, provided that the relations are first sorted in join-column order. It appears that the equivalence-transformation approach developed in this paper may be adopted as the foundation for an optimizer of SQL-like queries and the nested-iteration method may then be used to augment the performance of the optimizer for the rather special situations for which the latter method is more efficient.

This section analyzes and compares the costs of processing types N, J, JA, and D nested queries using the System R approach and the transformation algorithms presented in this paper. For simplicity, nested queries of depth greater than 1 are

not considered. R_i denotes the relation of the outer query block, R_j the relation indicated in the FROM clause of the inner block, and R_t the temporary relation obtained by an intermediate processing on R_j. P_k is the size (in pages) of relation R_k, and N_k the number of tuples in R_k. f_i denotes the fraction of the tuples of R_i that satisfy all simple predicates on R_i, and J_{kn} the number of distinct values in the C_n column of R_k. Further, it is assumed that a $(B - 1)$-way multiway merge sort is used, which requires $2 \cdot P \cdot \log_{B-1} P$ page I/Os to sort a relation R [3, 4]. This section takes page I/Os as the primary measure of performance of a query, and assumes, for expository simplicity, that R_i and R_j are sequentially scanned.

System R evaluates a type-N nested query by first evaluating the inner query block in order to reduce the nested predicate to an equivalent, simple predicate, and by then evaluating the outer query block. As long as R_t, the unary relation obtained by evaluating the inner query block, is small enough to fit into $B - 1$ pages of the main memory buffer, clearly the System R approach is optimal: R_j needs to be fetched only once to generate R_t; and also R_i once, one page at a time into the remaining buffer page, to compare against R_t. The System R method costs at most

$$P_j + P_i \quad \text{page fetches}$$

If, however, R_t is larger than $B - 1$ pages, this approach can cause serious thrashing, since R_t has to be fetched once for each tuple of R_i that satisfies all other simple predicates on R_i. Then it costs up to

$$P_j + P_t + P_i + f_i \cdot N_i \cdot P_t \quad \text{page I/Os,}$$

where the first two terms are the cost of generating R_t.

In contrast, if the type-N nested query is transformed to its canonical equivalent to merge join R_i and R_t, the total cost is

$$P_j + P_t + 2 \cdot P_t \cdot \log_{B-1} P_t + P_i + P_{t1} + 2 \cdot P_{t1} \cdot \log_{B-1} P_{t1} + P_t + P_{t1},$$

where the first three terms are the cost of generating R_t and removing duplicates from it, the next three terms are the cost of restricting and projecting R_i into R_{t1} and sorting it, and the last two terms are the cost of merge joining R_t and R_{t1}. The cost of removing duplicates from R_t is subsumed by the cost of merge-joining R_t with R_{t1}, since R_t needs to be sorted for the merge-join anyway. Note that the total cost expressed in the above formula may be further reduced since R_t may be reduced in size by removal of duplicates.

Intuitively, the nested-iteration method of processing a nested query of any type will tend to be efficient and will obviate the need to transform the query, if (1) P_t is "large" (which increases the cost of sorting R_t for the merge-join, thereby placing the transformation approach at a disadvantage) and (2) $f_i \cdot N_i$ is very "small" (around $2 \cdot \log_{B-1} P_t$), so that the nested-iteration method will not require R_t to be retrieved as many times as it is required to sort R_t. However, these represent a very small subset of the set of all possible query and database characteristics. The following example compares the performance of the two methods for a type-N nested query.

Example 14. Suppose $B = 6$, $P_i = 100$, $f_i \cdot N_i = 500$, $P_j = 100$, and $P_t = 20$. The nested-iteration method may cost 10,220 page fetches. If R_{t1}, the temporary

relation generated by reducing R_i, is 50 pages and a five-way merge-sort is used to sort R_{t1} and R_t, the transformation approach costs total 720 page I/Os.

Now let us consider the cost of processing nested queries in which the inner query block contains a join predicate $R_i.C_n = R_j.C_m$, that is, types J, JA, and D nested queries, by the nested-iteration method. The drawback of the nested-iteration method is that, by definition, it requires R_j to be retrieved potentially very many times: $f_i \cdot N_i$ times if there is no index on the C_m column of R_j, and up to $\min(f_i \cdot N_i, J_{jm})$ times if C_m is indexed. If the index on the C_m column is *clustered*, that is, if the tuples of R_j that share the same key value are stored physically "close" together, only $|P_j/J_{jm}|$ pages may need to be retrieved each time. If the index on C_m is not clustered, $\min(|N_j/J_{jm}|, P_j)$ pages must be fetched each time.

The total cost of processing types J, JA, or D nested queries by the transformation approach proposed in this paper consists simply of the cost of generating a temporary relation R_t and merge-joining it with R_i. For a type-J nested query, the cost of generating R_t is

$$P_j + P_t + 2 \cdot P_t \cdot \log_{B-1} P_t,$$

where the last term is the cost of removing duplicates from R_t. Since duplicates are removed from R_t by sorting it, only R_i (or, more likely, R_{t1}, obtained by restricting and projecting R_i) needs to be sorted for the merge-join.

Example 15. Let $P_i = P_j = 50$, $P_t = 20$, $B = 6$, and $f_i \cdot N_i = 500$. The nested-iteration approach costs 10,120 page fetches for a type-J nested query. If a five-way external merge-sort technique is used to sort R_t and R_i in join-column order and R_i is not reduced in size by restriction and projection, the transformation approach costs 550 page I/Os.

The penalty for transforming a type-JA nested query to an equivalent type-J nested query has been shown to be the cost of generating temporary relation R_t by evaluating a query with a GROUP BY clause on the relation(s) of the inner query block. System R uses one of two methods in implementing the GROUP BY construct [19]. One is to use an index on the GROUP BY column. The other is to sort the relation in GROUP BY column order. Therefore, the intermediate-processing penalty can be seen to be

$$\min(P_j, 2 \cdot P_{t2} \cdot \log_{B-1} P_{t2} + P_j) + P_t,$$

where R_{t2} is the reduced relation which results from restricting and projecting R_j. The reader may easily verify that R_t thus obtained is free of duplicates and is in join-column order; that is, R_t need not be sorted to be merge-joined with R_i.

Example 16. Consider Q_3 and Q_4 of Section 4. Let $P_i = 50$, $P_j = 30$, $P_t = 5$, $B = 6$, and $f_i \cdot N_i = 100$. The nested-iteration method of processing Q_3 is, in the worst case, $50 + 100 \cdot 30 = 3050$ page fetches. If a five-way merge-sort technique is used (to generate R_t and to merge-join R_i and R_t), it costs 560 page I/Os to sort R_i and R_j. The merge-join of R_i and R_t costs additional 55 page fetches. So Q_4 incurs 615 page fetches. Note that the cost may be even smaller if R_i is first reduced by restriction and projection and the resulting reduced relation is sorted for the merge-join.

Recall that a type-D nested query contains two inner query blocks, which reference R_j and R_k, respectively. Assume for expository simplicity that the query block on R_j has in its WHERE clause a join predicate $R_i.C_n = R_j.C_m$, where R_i is the relation of the outer query block. The query block on R_k is assumed to contain only simple predicates, so that it may be processed only once to yield a unary relation R_{t3}. Then the nested-iteration method of processing a type-D nested query requires, for each tuple of R_i that satisfies simple predicates on R_i, retrieval of not only R_j but also R_{t3}.

In order to transform a type-D nested query to a semantically equivalent type-N nested query, one of the two relations in the inner query block must be relationally divided by the other. The binary division of a relation of degree 2, $R_j(C_1, C_2)$, by a unary relation $R_k(C_2)$ yields a unary relation $R_t(C_1)$. The quotient R_t can then be merge-joined with R_i.

R_t can be obtained by grouping the dividend R_j by the values in the C_1 column, and extracting the C_1 values from each group of tuples that contain in the C_2 column *all* the values of R_k. Since the dividend needs to be grouped by the values of the quotient column, sorting of the dividend relation in quotient-column order, as well as the divisor relation, has been suggested as a method for computing relational division [20]. Therefore, the cost of generating R_t for a type-D nested query is

$$P_j + P_{t2} + 2 \cdot P_{t2} \cdot \log_{B-1} P_{t2} + P_k + P_{t3} + 2 \cdot P_{t3} \cdot \log_{B-1} P_{t3} + P_{t2} + P_{t3} + P_t + P_i + P_{t1} + 2 \cdot P_{t1} \cdot \log_{B-1} P_{t1} + P_{t1} + P_t,$$

where the first six terms are the cost of generating temporary relations R_{t2} and R_{t3} from the two inner relations of the division predicate, the next three terms are the cost of dividing R_{t2} by R_{t3} and generating the quotient R_t, and the last five terms are the cost of merge-joining R_t and R_{t1}. As was the case with a type-JA nested query, the quotient R_t is also free of duplicates and is in join-column order to be merge-joined with R_{t1}.

Example 17. Let $P_i = 50$, $P_j = P_{t2} = 30$, $p_t = 5$, $P_k = P_{t3} = 1$, $B = 4$, and $f_i \cdot N_i = 1000$. The nested-iteration method of processing a type-D nested query requires, in the worst case, $50 + 1000 \cdot 30 + 1 = 30051$ page fetches. If a three-way merge-sort technique is used to sort R_{t3} and R_{t2}, it costs 296 page I/Os to generate the quotient of dividing R_{t2} by R_{t3}. Further, it costs 400 page I/Os to sort R_i. So the cost of merge-joining R_i and R_t is $400 + 50 + 5 = 455$. The total cost of processing the type-D nested query by the algorithms of this paper is $296 + 455 = 751$ page I/Os.

7. PROCESSING A GENERAL NESTED QUERY

So far it has been assumed for simplicity that the WHERE clause of a query block contains only one nested or division predicate. In general, however, a query block may be nested to an arbitrary depth and contain an arbitrary number of any type of predicates. This section presents a coherent strategy for processing such a general query.

It has been shown that a type-N or type-J nested query may be transformed directly to an equivalent canonical query (Algorithm NEST-N-J). A type-JA

nested query may be converted to a type-J nested query once query blocks with an aggregate function in the SELECT clause have been evaluated (Algorithm NEST-JA(G)). And a type-D nested query can be transformed to its canonical form once the relational division operation it implies has been performed (Algorithm NEST-D). What this means is that a nested predicate which gives rise to a type-N, type-J, or type-JA nesting and a division predicate which results in a type-D nesting may each be replaced by a set of join predicates. Further, it was shown in Section 2 that a nested predicate which yields a type-A nesting must be replaced by a simple predicate. Algorithm NEST-G, which replaces all nested predicates and division predicates in the WHERE clause of the outermost block of a general nested query with a set of join predicates and simple predicates, emerges immediately from these observations. The algorithm is described below in terms of the query graph for a general query.

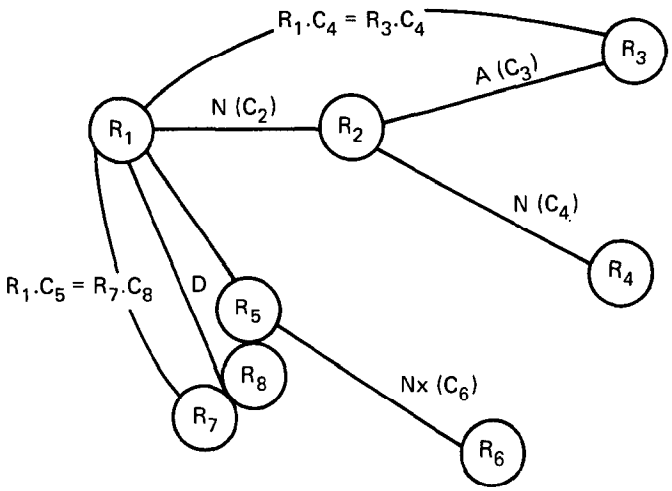
Algorithm NEST-G

(1) Transform each type-A predicate to a simple predicate by completely evaluating the query block Q represented by the right-hand node on the query graph for the subquery (predicate). If Q is itself nested, Algorithm NEST-G is invoked recursively on Q. Once Q has been evaluated, Q and the straight arc labeled 'A', leading to Q from the outermost query block, are eliminated from the initial query graph.
(2) Transform each type-JA nested subquery to an equivalent type-N or type-J subquery by Algorithm NEST-JA(G). If the right-hand node of the circular arc is further nested, Algorithm NEST-G may be invoked recursively on the node. The query graph for the resulting type-N or type-J nested subquery replaces the query graph for the initial type-JA subquery.
(3) Transform each type-D nested subquery to its canonical form by Algorithm NEST-D. If either of the two right-hand nodes on the query graph of the subquery is further nested, Algorithm NEST-G may be recursively invoked on the node. Once the division predicate has been replaced by an appropriate set of join predicates, the two right-hand nodes and both the straight arc and the circular arc leading to them are removed from the query graph of the initial subquery.
(4) Transform the resulting query, which consists only of type-N and type-J subqueries, to an equivalent canonical query by Algorithm NEST-N-J.

Example 18. This example illustrates Algorithm NEST-G. The following is a general nested query of considerable complexity.

```
SELECT   R1.C1
FROM     R1
WHERE    R1.C2  IS IN  (SELECT    R2.C2
                        FROM      R2
                        WHERE     R2.C3 = (SELECT    AGG(R3.C3)
                                           FROM      R3
                                           WHERE     R3.C4 = R1.C4)
                              AND  R2.C4  IS IN  (SELECT    R4.C4
                                                  FROM      R4))
  AND  R1.C3 = (SELECT    AGG(R5.C5)
                FROM      R5
                WHERE     R5.C6  IS NOT IN  (SELECT    R6.C6
                                             FROM      R6))
  AND  (SELECT    R7.C7
        FROM      R7
        WHERE     R7.C8 = R1.C5)
             =
       (SELECT    R8.C8
        FROM      R8);
```

The outermost block of the query consists of three predicates (subqueries); a type-JA nested predicate of depth 2, a type-A nested predicate of depth 2, and a type-D nested predicate. The Q of the type-JA nested predicate of depth 2 in turn consists of 2 predicates; a type-JA nested predicate of depth 1 and a type-N nested predicate of depth 1. The Q of the type-A nested predicate of depth 2 contains a type-N nested predicate of depth 1 involving the set noninclusion operator. The query graph for the above query is given below.



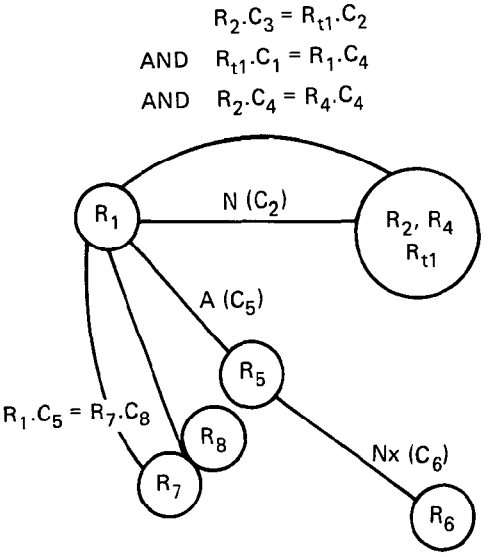
First, the type-JA subquery of depth 2 will be transformed to its type-J equivalent by Algorithm NEST-JA(G). The right-hand node of the subquery consists of two subqueries: one is type-JA of depth 1 and the other is type-N of depth 1. Algorithm NEST-JA transforms the type-JA nested predicate of depth 1 into a conjunction of join predicates, $R_2.C_3 = R_{t1}.C_2$ AND $R_{t1}.C_1 = R_1.C_4$, where the temporary relation R_{t1} is obtained by

```
Rt1(C1, C2) = (SELECT      R3.C4,  AGG(R3.C3)
               FROM        R3
               GROUP BY    R3.C4);
```

Further, Algorithm NEST-N-J transforms the type-N nested predicate of depth 1 into a join predicate, $R_2.C_4 = R_4.C_4$ Then the initial type-JA nested predicate of depth 2 will have been transformed to the following type-J nested predicate of depth 1 by Algorithm NEST-N-J.

```
R1.C2  IS IN  (SELECT    R2.C2
               FROM      R2,  R4,  Rt1
               WHERE     R2.C3 = Rt1.C2    AND
                         Rt1.C1 = R1.C4    AND
                         R2.C4 = R4.C4);
```

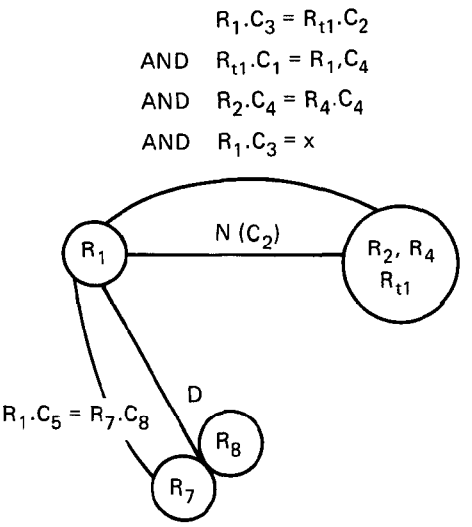
The following query graph has now been obtained.



Next, the type-A subquery of depth 2 is evaluated. The right-hand node of the subquery is itself type-N nested. A recursive call of Algorithm NEST-G transforms the node to

SELECT AGG($R_5.C_5$)
FROM R_5, R_6
WHERE $\neg(R_5.C_6 = R_6.C_6)$;

The transformed node is then evaluated to a constant x, and the nested predicate of the initial type-A subquery becomes a simple predicate, $R_1.C_3 = x$. The resulting query graph is shown below.



Then, the type-D subquery is evaluated so as to replace the division predicate with a join predicate by Algorithm NEST-D. The resulting join predicate is $R_1.C_5 = R_{t2}.C_1$, where the unary relation $R_{t2}(C_1)$ is the quotient of dividing $R_7(C_8, C_7)$ by $R_8(C_8)$.

The resulting query is type-J nested and Algorithm NEST-N-J can be used to transform it into its canonical equivalent, shown in the following.

```
SELECT   R1.C1
FROM     R1, Rt1, Rt2
WHERE    R1.C2 = R2.C2    AND
         R2.C3 = Rt1.C2   AND
         Rt1.C1 = R1.C4   AND
         R2.C4 = R4.C4    AND
         R1.C3 = x        AND
         R1.C5 = Rt2.C1;
```

8. SUMMARY

The fundamental structure of an SQL-like, block-structured data language has been analyzed. A query nested to an arbitrary depth has been shown to be composed of five basic types of nesting. Four of them have not been well understood and their present implementation suffers from the use of the inefficient nested-iteration method. Alternate ways of interpreting the operations of queries that involve these types of nesting have provided the basis for algorithms which transform the queries to equivalent, nonnested queries that existing optimizers are designed to process more efficiently. The algorithms have been shown to improve the performance of nested queries often by orders of magnitude. Finally, they have been combined into a coherent strategy for completely processing a general query of arbitrary complexity.

ACKNOWLEDGMENTS

I am grateful to Prof. David Kuck (Department of Computer Science, University of Illinois, Urbana) and Dr. Mario Schkolnick (IBM Research, San Jose) for their many valuable comments on the presentation and technical accuracy of this paper. I also thank the referees for their incisive comments about the treatment of duplicates and the analysis of the transformation algorithms in earlier versions of this paper, which helped to considerably improve the presentation of Sections 6 and 7.

REFERENCES

1. ASTRAHAN, M.M., AND CHAMBERLIN, D.D. Implementation of a structured English query language. *Commun. ACM 18*, 10(Oct. 1975), 580–588.
2. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2(June 1976), 97–137.
3. BLASGEN, M.W., AND ESWARAN, K.P. On the evaluation of queries in relational data base systems. IBM Res. Rep. RJ1745, IBM Research, San Jose, Calif., April 1976.
4. BLASGEN, M.W., AND ESWARAN, K.P. Storage and access in relational data bases. *IBM Syst. J. 16*, 4, 1977, 363–377.
5. BOYCE, R.F., AND CHAMBERLIN, D.D. SEQUEL: A structured English query language. In *Proc. ACM SIGMOD Workshop Data Description, Access and Control* (Ann Arbor, Mich., May 1–3), ACM, New York, 1974, pp. 249–264.
6. CHAMBERLIN, D.D., ET AL. SEQUEL2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev.* (Nov. 1976), 560–575.

7. CODD, E.F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6(June 1970), 377–387.
8. CODD, E.F. A database sublanguage founded on the relational calculus. In *Proc. ACM SIGFIDET Workshop on Data Description, Access and Control* (San Diego, Nov. 11–12), ACM, New York, 1971, pp. 35–68.
9. CODD, E.F. Further normalization of the data base relational model. In *Data Base Systems, Courant Computer Science Symposia,* Vol. 6, Prentice-Hall, Englewood Cliffs, N.J., 1971.
10. CODD, E.F. Relational completeness of data base sublanguages. In *Data Base Systems, Courant Computer Science Symposia,* Vol. 6, Prentice-Hall, Englewood Cliffs, N.J., 1971.
11. CZARNIK, B., SCHUSTER, S., AND TSICHRITZIS, D. ZETA: A relational data base management system. In *Proc. ACM Pacific Regional Conf.* (San Francisco, April 17–18), ACM, New York, 1975, pp. 21–25.
12. DATE, C.J. *An Introduction to Database Systems* (2nd ed.). Addison-Wesley, Reading, Mass., 1977.
13. EPSTEIN, R. Techniques for processing of aggregates in relational database systems. ERL/UCB Memo M79/8, Electronics Research Laboratory, Univ. California, Berkeley, Feb. 1979.
14. LIEN, Y.E. Design and implementation of a relational database on a minicomputer. In *Proc. ACM Annual Conf.* (Seattle, Oct. 16–19), ACM, New York, 1977, pp. 16–22.
15. MYLOPOULOS, J., SCHUSTER, S., AND TSICHRITZIS, D. A multi-level relational system. In *Proc. 1975 AFIPS Nat. Computer Conf.*, Vol. 44. AFIPS Press, Arlington, Va., pp. 403–408.
16. PALERMO, F.P. A data base search problem. IBM Res. Rep. RJ1072, San Jose, Calif., July 1972.
17. ROTHNIE, J.B. An approach to implementing a relational data base management system. In *Proc. ACM SIGMOD Workshop Data Description, Access and Control.* (Ann Arbor, Mich., May 1–3), ACM, New York, 1974, pp. 277–294.
18. ROTHNIE, J.B. Evaluating inter-entry retrieval expressions in a relational data base management system. In *Proc. 1975 AFIPS Nat. Computer Conf.*, Vol. 44. AFIPS Press, Arlington, Va., pp. 417–423.
19. SELINGER, P.G., ET AL. Access path selection in a relational database system. In *Proc. Inter. Conf. Management of Data* (ACM) (Boston, Mass., May 1979), 23–34.
20. SMITH, J.M., AND CHANG, P.Y. Optimizing the performance of a relational algebra database interface. *Commun. ACM 18*, 10(Oct. 1975), 568–579.
21. Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.
22. WEELDREYER, J.A., AND FRIESEN, O.D. Multics relational data store: An implementation of a relational data base manager. In Proc. 11th Hawaii Int. Conf. Systems Science, 1978, pp. 52–66.
23. WEISS, H.M. The ORACLE data base management system. *Mini-Micro Syst.* (Aug. 1980), 111–114.
24. WONG, E., AND YOUSSEFI, K. Decomposition—A strategy for query processing. *ACM Trans. Database Syst. 1*, 3(Sept. 1976), 223–241.

Received February 1981; revised May 1981; accepted July 1981