Lawrence Berkeley National Laboratory
Recent Work

Title
OPTIMIZATION OF NESTED SQL QUERIES REVISITED

Permalink
https://escholarship.org/uc/item/6cv5q5r9

Authors
Ganski, R.A.
Wong, H.K.T.

Publication Date
1987-04-01
OPTIMIZATION OF NESTED SQL QUERIES REVISITED

R.A. Ganski and H.K.T. Wong

April 1987

TWO-WEEK LOAN COPY

This is a Library Circulating Copy which may be borrowed for two weeks.

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098
DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
Optimization of Nested SQL Queries Revisited

Richard A. Ganski
Department of Computer Science
San Francisco State University

Harry K.T. Wong
Lawrence Berkeley Laboratory
University of California
Berkeley, California

Abstract

Current methods of evaluating nested queries in the SQL language can be inefficient in a variety of query and data base contexts. Previous research in the area of nested query optimization which sought methods of reducing evaluation costs is summarized, including a classification scheme for nested queries, algorithms designed to transform each type of query to a logically equivalent form which may then be evaluated more efficiently, and a description of a major bug in one of these algorithms. Further examination reveals another bug in the same algorithm. Solutions to these bugs are proposed and incorporated into a new transformation algorithm, and extensions are proposed which will allow the transformation algorithms to handle a larger class of predicates. A recursive algorithm for processing a general nested query is presented and the action of this algorithm is demonstrated. This algorithm can be used to transform any nested query.

1. Introduction

SQL is a block-structured query language for data retrieval and manipulation developed at the IBM Research Laboratory in San Jose, California [AST 75]. SQL was incorporated into System R, the relational data base management system, also developed at the IBM San Jose Research Laboratory [AST 76].

One of the most powerful features of SQL is the nesting of query blocks. For demonstration purposes, assume the following relations:

\[
\begin{align*}
S & : (SNO, SNAME, STATUS, CITY) & \text{the Suppliers relation} \\
P & : (PNO, PNAME, COLOR, WEIGHT, CITY) & \text{the Parts relation} \\
SP & : (SNO, PNO, QTY, ORIGIN) & \text{the Shipments relation}
\end{align*}
\]

The primary keys for these relations are SNO, PNO, and SNO,PNO respectively. If we wanted the names of all suppliers who supply part P2 we could say:

\[
\begin{align*}
\text{SELECT} & \quad \text{SNAME} \\
\text{FROM} & \quad S \\
\text{WHERE} & \quad \text{SNO IN} (\text{SELECT} \quad \text{SNO} \\
& \quad \text{FROM} \quad SP \\
& \quad \text{WHERE} \quad \text{NO = 'P2'});
\end{align*}
\]

This is an example of a query with a single level of nesting. The basic structure of a SQL query is a query block, which consists principally of a SELECT clause, a FROM clause, and zero or more WHERE clauses. The first query block in a nested query is known as the outer query block and the next query block is known as the inner query block. The WHERE clause specifies the predicates which the tuples retrieved must satisfy. One type of predicate which can appear in the WHERE clause is a nested predicate, which is of the form \([R_i.\text{Col} \text{ op } Q]\), where \(Q\) is a query block \([\text{KIM 82:445}]\). \(Q\) will always be a form of the SELECT statement. The \(\text{op}\) may be a scalar or a set membership operator. A relation referred to in the inner query block shall be designated as an inner relation, and a relation referred to in the outer query block shall be designated as an outer relation. Queries can be nested to an arbitrary depth.

In his 1982 paper “On Optimizing an SQL-like Nested Query” [KIM 82], Won Kim showed that the conventional techniques used in implementing query nesting, i.e. the techniques used in System R [SEL 79:33], can be very inefficient: tables referenced in the inner query block of a nested query may have to be retrieved once for each tuple of the relation referenced in the outer query block [KIM 82:450]. As a solution to this problem, Kim proposed query transformation algorithms that would improve the efficiency of nested query evaluation, sometimes by orders of magnitude. His approach was to transform a nested query to a logically equivalent single-level query (i.e. without nesting): this query could then be examined by a query optimizer, such as that described in [SEL 79], for alternative methods of processing, including different methods of performing joins. To introduce Kim's results, his system of classification for nested queries is outlined below.

2. Types of Nested Queries

Won Kim developed a classification of nested query types, four of which are relevant to this paper. They are described here briefly for single-level nested queries, as presented in [KIM 82].
2.1. Type-A Nesting

A nested predicate is type-A if the inner query block Q does not contain a join predicate that references a relation in the outer query block, and if the SELECT clause of Q consists of an aggregate function over a column in an inner relation [KIM 82:446]. The following is an example of a type-A nested query of depth one:

\[
\begin{align*}
\text{SELECT} & \quad \text{SNO} \\
\text{FROM} & \quad \text{SP} \\
\text{WHERE} & \quad \text{PNO} = \text{(SELECT MAX(PNO))} \\
& \quad \text{FROM} \quad P;
\end{align*}
\]

Since the inner query block of a type-A nested query does not reference a relation of the outer query block, it may be evaluated independently of the outer query block, and the result of its evaluation will be a single constant [SEL 79:33].

2.2. Type-N Nesting

A nested predicate is type-N if the inner query block Q does not contain a join predicate which references a relation in the outer block, and the SELECT clause of Q does not contain an aggregate function [KIM 82:447]. The following is an example of a type-N nested query:

\[
\begin{align*}
\text{SELECT} & \quad \text{SNO} \\
\text{FROM} & \quad \text{SP} \\
\text{WHERE} & \quad \text{PNO} \text{ IS IN} \text{ (SELECT PNO)} \\
& \quad \text{FROM} \quad P \quad \text{WHERE} \quad \text{WEIGHT} > 50;
\end{align*}
\]

Evaluation of a Type-N Nested Query. This kind of nested query would be processed in System R by first processing the inner query block Q, resulting in a list of values X which can then be substituted for the inner query block in the nested predicate, so that \text{PNO IS IN Q} becomes \text{PNO IS IN X}. The resulting query is then evaluated by nested iteration [SEL 79:33].

2.3. Type-J Nesting

A type-J nested predicate results when the WHERE clause of the inner query block Q contains a join predicate which references the relation of an outer query block, and the relation is not mentioned in the inner FROM clause. Another condition is that the SELECT clause of the inner query block does not contain an aggregate function [KIM 82:448]. The following is an example of type-J nesting:

\[
\begin{align*}
\text{SELECT} & \quad \text{SNAME} \\
\text{FROM} & \quad S \\
\text{WHERE} & \quad \text{SNO} \text{ IS IN} \text{ (SELECT SNO)} \\
& \quad \text{FROM} \quad SP \quad \text{WHERE} \quad \text{QTY} > 100 \text{ AND} \quad \text{SPORIGIN} = \text{S.CITY};
\end{align*}
\]

2.4. Type-JA Nesting

Type-JA nesting is present when the WHERE clause of the inner query block contains a join predicate which references the relation of an outer query block, and the inner SELECT clause consists of an aggregate function over an inner relation [KIM 82:449]:

- Select names of parts which have the highest part number in the city from which they are supplied.

\[
\begin{align*}
\text{SELECT} & \quad \text{PNAME} \\
\text{FROM} & \quad P \\
\text{WHERE} & \quad \text{PNO} = \text{(SELECT MAX(PNO))} \\
& \quad \text{FROM} \quad SP \quad \text{WHERE} \quad \text{SPORIGIN} = \text{P.CITY};
\end{align*}
\]

Evaluation of Type-J and Type-JA Nested Queries. Type-J and type-JA nesting are processed in System R by the nested iteration method: the inner query block is processed once for each tuple of the outer relation which satisfies all simple predicates on the outer relation [SEL 79:33]. This method has the obvious disadvantage that the inner relation (SP in example 4) may have to be retrieved many times: in example 4, it must be retrieved once for each tuple of the outer relation S, since there are no simple predicates in the outer query block. It is this inefficiency which motivated Kim to develop alternative algorithms for processing nested queries.

3. Kim's Algorithms for Processing Nested Queries

Kim observed that for type-N and type-J nested queries, the nested iteration method for processing nested queries is equivalent to performing a join between the outer and inner relations [KIM 82:451]. But nested iteration is only one way of performing a join; for single-level queries System R also performs joins by the merge join method, with the decision as to which method to use made by the query optimizer [SEL 79:28]. Kim showed that nested queries could be transformed to logically equivalent single-level queries containing single-level join predicates explicitly, and that now the query optimizer can choose a merge join method in implementing the joins, often at a great reduction of cost over the nested iteration method [KIM 82:461]. Kim's transformation algorithms are summarized in the present section.

3.1. Processing a Type-N or Type-J Nested Query

In his Lemma 1 [KIM 82:451], Kim states that a type-N nested two-relation query is equivalent to a canonical two-relation query with a join predicate:

Let Q1 be

\[
\begin{align*}
\text{SELECT} & \quad \text{R1.Ck} \\
\text{FROM} & \quad \text{R1.Rj} \\
\text{WHERE} & \quad \text{R1.Ch} = \text{Rj.Cm};
\end{align*}
\]

and let Q2 be

\[
\begin{align*}
\text{SELECT} & \quad \text{R1.Ck} \\
\text{FROM} & \quad \text{Ri} \\
\text{WHERE} & \quad \text{R1.Ch IS IN} \text{ (SELECT Rj.Cm)} \\
& \quad \text{FROM} \quad \text{Rj});
\end{align*}
\]

(KIM 82:451)

Kim's Lemma 1 states that Q1 and Q2 are equivalent; that is,
they yield the same result [KIM 82:451]. Kim's proof of lemma 1 calls attention to the fact that by definition the inner block of Q2 can be evaluated independently of the outer block, resulting in a list of values. Since this list contains values from column Rj.Cm, the predicate is equivalent to the join predicate Rj.Ch = Rj.Cm [KIM 82:451-452]. From Lemma 1 Kim develops the following algorithm:

Algorithm NEST-N-J
1. Combine the FROM clauses of all query blocks into one FROM clause.
2. AND together the WHERE clauses of all query blocks, replacing IS IN by =.
3. Retain the SELECT clause of the outermost query block.

The result is a canonical query logically equivalent to the original nested query. The algorithm applies to type-N or type-J nested queries with one or more levels of nesting.

3.2. Processing a Type-JA Nested Query

In his Lemma 2 [KIM 82:455], Kim asserts that a type-JA nested query can be transformed to a type-J nested query which references a new temporary relation:

Let Q3 be

\[
\text{SELECT } R.Ck \\
\text{FROM } R \\
\text{WHERE } R.Ch \neq (\text{SELECT } \text{AGG}(Rj.Cm) \\
\text{FROM } Rj \\
\text{WHERE } Rj.Cn = R.Cp);
\]

and let Q4 be

\[
\text{SELECT } R.Ck \\
\text{FROM } R \\
\text{WHERE } R.Ch \neq (\text{SELECT } R.C2 \\
\text{FROM } R \\
\text{WHERE } R.C1 = R.Cp);
\]

where R is a temporary table obtained by

\[
\text{SELECT } R(C1.C2) = (\text{SELECT } Rj.Ch, \text{AGG}(Rj.Cm) \\
\text{FROM } Rj \\
\text{GROUP BY } Rj.Ch);
\]

[KIM 82:454-455]

Kim's Lemma 2 states that Q3 and Q4 are equivalent [KIM 82:455]. His proof postulates that the action of the nested iteration processing of a type-JA query can be captured in a temporary table formed with a GROUP BY clause, as in R: for each tuple of R, a tuple is retrieved from R whose C1 (formerly C) value matches the Cp value of the Ri tuple. The C2 value of the Ri tuple will contain the aggregate value obtained by the GROUP BY clause, and this can be matched with R.Ch.

[KIM 82:455]

Lemma 2 leads to an algorithm which transforms a type-JA nested query of depth one to an equivalent type-J nested query of depth 1. Assume a type-JA nested query as follows:

\[
\text{SELECT } R1.Cn+2 \\
\text{FROM } R1 \\
\text{WHERE } R1.Cn+1 = (\text{SELECT } \text{AGG}(R2.Cn+1) \\
\text{FROM } R2 \\
\text{WHERE } R2.C1 = R1.C1 \text{ AND} \\
\text{R2.C2 = R1.C2})
\]

[KIM 82:455]

Algorithm NEST-JA
1. Generate a temporary relation R(C1,...,Cn,Cn+1) from R2 such that R.Cn+1 is the result of applying the aggregate function AGG on the Cn+1 column of R2 which have matching values in R1 for C1,C2, etc.
2. Transform the inner query block of the initial query by changing all references to R2 columns in join predicates which also reference R1 to the corresponding R1 columns. The result is a type-J nested query, which can be passed to algorithm NEST-N-J for transformation to its canonical equivalent.

[KIM 82:455-456]


Kim's analyses of his algorithms [KIM 82:461-464] compare the costs of processing N, J, and JA-type nested queries using the nested iteration method and the transformation method followed by merge joins. Kim develops cost functions for each method and for each type of nesting, using variables such as the sizes of relations, available memory buffer space, and selectivity factors. He demonstrates the cost reductions attainable by his transformation method with examples of queries and data base conditions for each type of nesting. The following table summarizes the results Kim obtained in three of his examples [KIM 82:462-463]:

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Nested Iteration</th>
<th>Merge Join (Page I/O's)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-N</td>
<td>10,220</td>
<td>720</td>
</tr>
<tr>
<td>Type-J</td>
<td>10,120</td>
<td>550</td>
</tr>
<tr>
<td>Type-JA</td>
<td>3,050</td>
<td>615</td>
</tr>
</tbody>
</table>

Figure 1: Page I/O’s Required in Kim's Examples

The comparative costs will of course vary with different queries and data base conditions, but Kim has shown that cost savings of 80% to 95% are possible with his transformation method.

5. Bugs in Kim's Algorithm NEST-JA and their Solutions

5.1. The COUNT bug

In a 1984 U.C. Berkeley Memorandum [KIE 84], Werner Kiessling revealed a problem with Kim's algorithm NEST-JA. The problem arises when a type-JA nested query contains the COUNT function. To illustrate his arguments, Kiessling defines two relations:
The following instantiations of these relations are assumed:

<table>
<thead>
<tr>
<th>PARTS:</th>
<th>SUPPLY:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNUM</td>
<td>QUAN</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kiessling defines Query Q2 as follows:

Query Q2:

Find the part numbers of those parts whose quantities on hand equal the number of shipments of those parts before 1-1-80:

\[
\text{SELECT PNUM FROM PARTS WHERE QOH = (SELECT COUNT(SHIPDATE) FROM SUPPLY WHERE SUPPLY.PNUM = PARTS.PNUM AND SHIPDATE < 1-1-80)}
\]

Given the example tables PARTS and SUPPLY defined above, query Q2 will give the following result when evaluated using nested iteration:

Result: PARTS PNUM
10
8

Application of Kim's algorithm NEST-JA to Query Q2 results in the following transformation:

\[
\text{TEMP'}(\text{SUPPNUM, CT}) =
\text{(SELECT PNUM, COUNT(SHIPDATE) FROM SUPPLY WHERE SHIPDATE < 1-1-80 GROUP BY PNUM)}
\]

\[
\text{SELECT PNUM FROM PARTS, TEMP'} \text{ WHERE PARTS.QOH = TEMP'.CT AND PARTS.PNUM = TEMP'.SUPPNUM}
\]

\[
\text{TEMP'} \text{ evaluates to}
\]

\[
\text{TEMP'}: \text{SUPPNUM CT}
\]

3 2
10 1

and the final result is

This result differs from that obtained using nested iteration. The reason why the transformation fails is that in the formation of the temporary relation, no tuples appear which do not match the predicates applied to the inner relation. Thus, the COUNT function will never return zero, since the only groups it is applied to are groups of tuples matching the predicates. Thus CT in the temporary relation will never be zero.

Kiessling explored a trial correction of the bug which involved ORing a predicate to the WHERE clause of the transformed query in order to a posteriori find where an empty set occurs to satisfy the predicate, but the trial correction failed on a query with more than one level of nesting [KIE 84:5]. Kiessling concludes that in attempting to use Kim's algorithm NEST-JA for transforming type-JA nested queries, "...there seems to be no general way to recover values lost by COUNTs on a correlation level greater than 1." [KIE 84:7]. While this does seem to be true in the context of the SQL language as specified in [AST 76], the problem can be solved if the outer join operation is available in the processing of the query.

5.2. Solution to the COUNT bug using outer joins

If either internally or through extensions to the query language an outer join operation may be specified as the join operation, the COUNT bug can be solved by performing an outer join in the creation of the temporary relation. The operation of outer join is defined in [COD 79:407]: the outer join includes all values from columns participating in join, with NULLs in the opposite column if there is no match for a column value. For example, assume the following relations:

\[
\begin{align*}
\text{R:} & \quad X \quad S: \quad X \\
& \quad A \quad B \\
& \quad B \quad C \\
& \quad E \\
\end{align*}
\]

An outer join between R and S, which will be designated R.X \# S.Y will have the following result:

\[
\begin{align*}
X & \quad ^{\wedge}\quad X \\
A & \quad ^{\wedge}\quad A \\
B & \quad ^{\wedge}\quad B \\
C & \quad ^{\wedge}\quad C \\
E & \quad ^{\wedge}\quad E \\
\end{align*}
\]

where ^ is the special null value. The outer join operation is implemented in at least one commercial data base management system with which the authors are familiar [ORA 86].

To solve the COUNT bug an outer join may be used in the creation of the temporary relation. Kiessling's query Q2 could be transformed to give the following:
$\text{TEMP3} (\text{SUPPNUM} . \text{CT}) =$

(\text{SELECT PARTS.PNUM, COUNT(SUPPLY.SHIPDATE) FROM PARTS, SUPPLY WHERE SUPPLY.SHIPDATE < 1-1-80 AND PARTS.PNUM = SUPPLY.PNUM GROUP BY PARTS.PNUM}) ;

Query T3:

SELECT PNUM FROM PARTS.1n1P3 WHERE PARTS.QOH = TEMP3.CT AND PARTS.PNUM = TEMP3.SUPPNUM;

Before looking at the result of this new query, let us look at the result of the outer join between PARTS and SUPPLY with the conditions given in the creation of the temporary relation TEMP3:

<table>
<thead>
<tr>
<th>PARTS.PNUM</th>
<th>PARTS.QOH</th>
<th>SUPPLY.PNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUPPLY.QUAN</th>
<th>SUPPLY.SHIPDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7-3-79</td>
</tr>
<tr>
<td>2</td>
<td>10-1-78</td>
</tr>
<tr>
<td>1</td>
<td>6-8-78</td>
</tr>
<tr>
<td>^</td>
<td>^</td>
</tr>
</tbody>
</table>

Note that the condition which applies to only one relation (SUPPLY.SHIPDATE < 1-1-80) must be applied before the join is performed. Otherwise the join would not contain the last row, and the result would be incorrect. This may happen if the join is performed first to take advantage of indices on the join columns. To ensure restriction, we can explicitly build a temporary table applying simple predicates. This temporary table will be a restriction and projection of the inner table:

$\text{TEMP2 (PNUM) = (SELECT PNUM FROM SUPPLY WHERE SHIPDATE < 1-1-80);}$

and TEMP3 is changed to

$\text{TEMP3 (SUPPNUM, CT) =}$

(\text{SELECT PARTS.PNUM, COUNT(TEMP2.SHIPDATE) FROM PARTS, TEMP2 WHERE PARTS.PNUM = TEMP2.PNUM GROUP BY PARTS.PNUM}) ;

Thus, TEMP3 will look like this:

$\text{TEMP3: SUPPNUM CT}$

<table>
<thead>
<tr>
<th>SUPPNUM</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

and the result of query T3 will be:

<table>
<thead>
<tr>
<th>PARTS.PNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

which matches the result obtained by nested iteration. This solution has been tested successfully on queries with more than a single level of nesting, including Kiessling’s query Q3 [KIE 84:6].

If the type-JA query with a COUNT function contains a nested join predicate with a scalar comparison operator other than equality, the correct result is obtained if the scalar operator is used in the outer join operation to create the temporary relation and the join predicate in the original query is changed to equality.

5.2.1. Query Blocks with COUNT(*)

If the SELECT clause of the inner query block contains COUNT(*) instead of COUNT(column name) then this approach must be modified. For example, if query Q2 contained a COUNT(*) instead of a COUNT(SHIPDATE), then the temporary table would look like this:

$\text{TEMP3: SUPPNUM CT}$

<table>
<thead>
<tr>
<th>SUPPNUM</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

This would be semantically incorrect, and the final result would be incorrect. To avoid this error, the SELECT clause used in the creation of the table must contain COUNT(col-name) instead of COUNT(*), where col-name is the name of some column in the inner relation. Since the join column of the inner relation will always be present in the original query and may be the only one that is, let col-name be the name of the join column of the inner relation. In our example it would be COUNT(TEMP2.PNUM).

5.3. Another Bug: Relations other than Equality

For aggregate functions other than COUNT Kim’s algorithm NEST-JA works correctly for nested join predicates containing the equality operator. However, if we consider other operators, we discover another bug in Kim’s algorithm.

Assume the PARTS and SUPPLY tables:

<table>
<thead>
<tr>
<th>PARTS:</th>
<th>SUPPLY:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNUM</td>
<td>QOH</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and the following type-JA query:

$\text{PARTS.PNUM}$

$\text{10}$

$\text{8}$

This would be semantically incorrect, and the final result would be incorrect. To avoid this error the SELECT clause used in the creation of the table must contain COUNT(col-name) instead of COUNT(*), where col-name is the name of some column in the inner relation. Since the join column of the inner relation will always be present in the original query and may be the only one that is, let col-name be the name of the join column of the inner relation. In our example it would be COUNT(TEMP2.PNUM).
Query Q5:

```
SELECT PNUM
FROM PARTS
WHERE QOH = (SELECT MAX(QUAN)
              FROM SUPPLY
              WHERE SUPPLY.PNUM < PARTS.PNUM AND SHIPDATE < 1-1-80);
```

This is the same as Kiessling's query Q1 [KIE 84:1] except for the substitution of the "<" operator for "=" operator in the join predicate. The result according to nested iteration semantics, assuming MAX(()) = NULL, is

```
PARTS PNUM
8
```

Kim's algorithm results in the following temporary table and transformed query:

```
TEMPS (SUPPNUM, MAXQUAN) = SELECT PNUM, MAX(QUAN)
FROM SUPPLY
WHERE SHIPDATE < 1-1-80
GROUP BY PNUM;
```

Query T5:

```
SELECT PNUM
FROM PARTS, TEMPS
WHERE QOH = TEMPS.MAXQUAN AND TEMPS.SUPPNUM < PARTS.PNUM;
```

and the following results:

```
TEMPS:
SUPPNUM  MAXQUAN
3        4
10       1
9
```

which does not match the results obtained by nested iteration. The problem is that the temporary table created by Kim's algorithm contains only aggregate information about tuples with the same join column value, whereas query Q5 asks for aggregate information about a range of join column values.

5.3.1 Solution to the Relations-other-than-Equality Bug

The solution to this bug is similar to the solution to the COUNT bug: perform a join in the creation of the temporary relation, only this time it need not be an outer join, unless the aggregate function is COUNT. The join in effect causes the temporary table to include aggregate values over the proper range of join column values. As before, the join predicate in the original query must be changed to equality. This implies that only the equality operator may be the outer relation and the temporary relation.

If this solution is applied to query Q5 and the last SUPPLY table, the outcome is:

```
TEMPS (SUPPNUM, MAXQUAN) =
SELECT PARTS.PNUM, MAX(SUPPLY.QUAN)
FROM PARTS, SUPPLY
WHERE SHIPDATE < 1-1-80 AND SUPPLY.PNUM < PARTS.PNUM
GROUP BY PARTS.PNUM;
```

and query Q5 is transformed to

Query T6:

```
SELECT PNUM
FROM PARTS, TEMPS
WHERE PARTS.QOH = TEMPS.MAXQUAN AND PARTS.PNUM = TEMPS.SUPPNUM;
```

with the following results:

```
TEMPS:
SUPPNUM  MAXQUAN
10       5
8        4
```

This matches the result obtained by nested iteration.

5.4. A Problem with Duplicates

The methods outlined above to solve the COUNT bug work correctly if the outer relation of the nested query contains no duplicates in the join column, but a problem arises if it does contain duplicates. Assume the following PARTS and SUPPLY relations:

```
PARTS: OOH
1 10 8
10 10 6
10 10 2
9 3 4
```

For this example let us again assume Kiessling's query Q2. If we apply query Q2 to the above relations, the result by nested iteration would be:

```
PARTS PNUM
3 10 8
```

If we apply our new modified version of Kim's algorithm, the results would be:

```
TEMPS: SUPPNUM  CT
3 4
10 2
8 0
```

This does not match the result obtained by nested iteration. The problem arises because duplicates in the outer relation increase the COUNT over that column in the temporary relation. This
problem does not arise with the MAX and MIN functions, but it does arise with the COUNT, AVG and SUM functions.

5.4.1. Solution to the Duplicates Problem

In order to match the results obtained by nested iteration semantics for relations with duplicates in the outer join column, our algorithm must be modified to remove duplicates before the join in the creation of the temporary table is performed. This can be accomplished by projecting the join column of the outer relation, and using the projection instead of the outer relation in any join required to build a temporary table. This is part of the procedure followed in INGRES [STO 76] for nested QUEL queries [KIE 84:8]. The efficiency of the algorithm can be improved by applying all simple predicates to the outer relation in the creation of the projection. In query Q2 this rule will have no effect since there are no simple predicates in the outer query block.

Using Kiessling’s query Q2 as an example again, let TEMP1 be defined as follows:

\[
\text{TEMP1(PNUM)} = (\text{SELECT DISTINCT PNUM FROM PARTS})
\]

TEMP1 is the projection of the PNUM column from PARTS. TEMP3 will now be defined as:

\[
\text{TEMP3(SUPPNUM,CT)} = (\text{SELECT TEMP1.PNUM, COUNT(SUPPLY.SHIPDATE) FROM TEMP1,SUPPLY WHERE SUPPLY.SHIPDATE < 1-1-80 AND TEMP1.PNUM = SUPPLY.PNUM GROUP BY TEMP1.PNUM})
\]

and query T3 remains the same. The results are:

<table>
<thead>
<tr>
<th>TEMP1:</th>
<th>TEMP3:</th>
<th>Final result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNUM:</td>
<td>SUPPNUM</td>
<td>CT</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

which matches the result obtained by nested iteration.

6. Modified Algorithm NEST-JA2

6.1 The Algorithm

The solutions to the bugs described in the previous section suggest a modified algorithm for transforming type-JA nested queries, which shall be called algorithm NEST-JA2. This algorithm consists of three major parts:

Algorithm NEST-JA2

1. Project the join column of the outer relation, and restrict it with any simple predicates applying to the outer relation.
2. Create a temporary relation, joining the inner relation with the projection of the outer relation. If the aggregate function is COUNT, the join must be an outer join, and the inner relation must be restricted and projected before the join is performed. If the aggregate function is COUNT(*), compute the COUNT function over the join column. The join predicate must use the same operator as the join predicate in the original query (except that it must be converted to the corresponding outer operator in the case of COUNT), and the join predicate in the original query must be changed to =. In the SELECT clause, select the join column from the outer table in the join predicate instead of the inner table. The GROUP BY clause will also contain columns from the outer relation.
3. Join the outer relation with the temporary relation, according to the transformed version of the original query.

To illustrate the action of algorithm NEST-JA2, let us apply it to Kiessling’s query Q2. The three steps are then as follows:

1. TEMP1 (PNUM) = SELECT DISTINCT PNUM FROM PARTS;
2. TEMP2 (PNUM) = (SELECT PNUM FROM SUPPLY WHERE SHIPDATE < 1-1-80);
3. TEMP3 (PNUM,CT) = (SELECT TEMP1.PNUM, COUNT(TMP2.SHIPDATE) FROM TEMP1, TEMP2 WHERE TEMP1.PNUM = TEMP2.PNUM GROUP BY TEMP1.PNUM);

If these three steps are applied to the PARTS and SUPPLY relations with duplicates considered above, the results are:

<table>
<thead>
<tr>
<th>TEMP1:</th>
<th>TEMP3:</th>
<th>Final result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNUM:</td>
<td>SUPPNUM</td>
<td>CT</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

which matches the result obtained by nested iteration.

7. Analysis of Modified Algorithm NEST-JA2

The total cost of processing a type-JA nested query using the new algorithm NEST-JA2 will consist of three major sub-costs:

1. The projection and restriction of the outer table Ri, resulting in temporary table R12.
2. The creation of temporary relation R1 by projecting and restricting inner relation Rj, joining this with temporary table R12, and performing a GROUP BY operation on the result.
3. Joining temporary table R1 with outer table Ri.

These costs will be examined in detail below. For simplicity it will be assumed that nested queries are of depth one. The analyses will be presented using Kim’s notation [KIM 82:462]: Ri denotes the relation of the outer query block, Rj the relation in the FROM clause of the inner query block, and Rk the temporary relation obtained by intermediate processing on Rj. Pk
is the size in pages of relation Rk, and Nk is the number of
tuples in Rk. Let f(i) denote the fraction of the tuples of Rk
that satisfy all simple predicates on Ri. B denotes the size in pages
of available main-memory buffer space. When it is necessary
to sort a relation, a (B-1)-way multi-way merge sort is used,
which requires 2*B*log_b(Pi) page I/O's to sort a relation R
[Kim 82:462]. The measure of performance is the number of
disk page I/O's required, and for simplicity relations Ri and Rj
are scanned sequentially.

7.1. Projection and Restriction of the Outer Table

The cost of creating a projection and restriction Rk2 from Ri,
with duplicates removed, is

\[ P_i + P_2 + 2*N_2*log_b(P_2) \text{ page I/O's} \]

where the last term is the cost of removing duplicates using a
(B-1)-way merge sort. This also sets up Rk2 in join column
order for a merge join. P2 will be some fraction of P1. Since Rk2
contains only tuples satisfying the simple predicates on Ri, P2
will be some fraction of f(i)*Pi, the fraction depending on the
size of the column compared to the size of a tuple.

7.2. Creation of Temporary Table Rk

In the modified algorithm NEST-JA2, a join is required in
the creation of the temporary relation from the inner relation. If
the aggregate function in the inner block is COUNT(), this join
will be an outer join. The inner relation is denoted Ri and Rk3
will designate a temporary relation created by projecting and
restricting Rj. Rk3 is used to perform the join with Rk2, followed
by the GROUP BY operation, to create the temporary
relation Rk.

The cost of this join will depend on whether the nested iterated
or the merge join method is used. The nested loops method
will be efficient if the temporary relation Rk3 can fit into B-1
memory pages, with a cost of

\[ P_j + P_2 + P_4 \text{ page I/O's} \]

where Rk4 is the result of the join. If, however, Rk3 does not fit
into B-1 pages, Rk3 will have to be retrieved once for each
tuple of Rk2, since Rk2 has already been restricted. The cost will be

\[ P_j + P_3 + P_2 + N_2*N_3 + P_4 \text{ page I/O's} \]

where the first two terms are the cost of creating Rk3.
If the merge join method is used, the cost will be

\[ P_j + P_3 + 2*N_3*log_b(P_3) + P_2 + P_3 + P_4 \text{ page I/O's} \]

where the first three terms are the cost of building Rk3, sorting
it and removing duplicates, and the last three terms are the cost
of merge joining Rk2 with Rk3 and storing the result. The cost
of sorting Rk2 is not included in the merge join cost, since this
cost is subsumed by the cost of creating it with duplicates
removed. In addition, performing a merge join to create Rk4
obviates the need to sort it for the GROUP BY operation, since
the GROUP BY column is the join column.

If the aggregate function in the inner SELECT clause is
COUNT(), an outer join must be used in the creation of tem-
porary table Rk4. The merge join method of performing an
outer join will have a cost function identical to that for a stan-
dard join, since the two relations are scanned in sorted order,
and no extra cost is involved in determining which tuples have
no matching tuples in the opposite relation. Rk4, the result of
the join, may be slightly larger than if a standard join were
performed, adding a small amount to the cost of the join. As in
Kim's analyses, the joins performed following transformation
will be assumed to be merge joins.

7.3. Join of Rt and Ri

The cost of joining temporary table Rt and outer table Ri
will also depend on the kind of join used, but as will be seen
below, a merge join of these relations can be particularly effi-
cient, since Rt is already in join column order: a merge join
will cost

\[ 2*Nt*log_b(Pt) + Pi + Pt \text{ page fetches} \]

assuming Ri is not reduced in size, while a nested iteration join
would cost

\[ Pi + f(i)*Nt*Pt \text{ page fetches} \]

7.4. Total Cost

The total cost of processing a single-level type-JA nested
query using the modified algorithm NEST-JA2 will depend on
the type of join used to create temporary relation Rk4 as shown
above; it will also depend on the type of join used between the
outer relation Ri and the temporary relation Rk. Thus there are
four possible total costs for a single-level query, each of which
may be estimated by the optimizer. One of these evaluation
methods in particular is worthy of note: the use of two merge
joins in the evaluation of the query. In evaluating the query by
this method there will be cost savings in the merge joins from
sorting relations earlier in the process: Rk2 is created in join
column order, so it does not have to be sorted for the join with
Rk3; Rk4 is created in GROUP BY column order, so it does not
have to be sorted for the GROUP BY operation; and Rt is
created in join column order, so it does not have to be sorted
for the merge join with Ri. The total cost for this method is

\[ P_i + P_2 + 2*N_2*log_b(P_2) + P_2 +
\]

\[ P_j + P_3 + 2*N_3*log_b(P_3) + P_2 + P_3 + 2*N_4 + P_t +
\]

\[ 2*Nt*log_b(P_t) + Pi + Pt \]

assuming Ri is not reduced in size, and where the first three
terms are the cost of projecting and restricting Ri, resulting in
Rk2; the next eight terms are the cost of creating temporary
table Rt, including the GROUP BY operation; and the last three
terms are the cost of performing the final join.

The modified algorithm can be compared to the nested iterated
method in the following example. Let the query to be eval-
uated be Kim's query Q3 [Kim 82:454] where the ag-
gregate function is MAX(). Let Pi = 50, Pj = 30, P2 = 7, P3 =
10, P4 = 8, Pt = 5, B = 6, and f(i)*Ni = 100. The nested iterated
method of processing Q3 costs 3050 page fetches in the
worst case. The transformation approach, using the modified algorithm and two merge joins, costs about 475 page fetches.

8. Extensions: the Predicates EXISTS, NOT EXISTS, ANY, and ALL

In presenting his transformation algorithms, Kim considered nested predicates containing scalar and set inclusion operators. If the language is extended to include the useful operators EXISTS, ANY, and ALL, some extensions to the transformation algorithms must be implemented. The extensions proposed in this section are transformations of the predicates to predicates containing simple scalar or set containment operators. The query can then be processed by the transformation algorithms presented above.

8.1 EXISTS and NOT EXISTS

A nested predicate of the form

WHERE EXISTS (SELECT selitems FROM fromitems WHERE whereitems)

can be transformed to the semantically equivalent nested predicate

WHERE 0 < (SELECT COUNT (selitems) FROM fromitems WHERE whereitems)

Similarly, a nested predicate of the form

WHERE NOT EXISTS (SELECT selitems FROM fromitems WHERE whereitems)

is transformed to the semantically equivalent predicate

WHERE 0 = (SELECT COUNT (selitems) FROM fromitems WHERE whereitems)

The resulting predicate is then processed as a type-A or type-JA predicate, depending on the details of the inner query block.

8.2 ANY and ALL

A predicate of the form

< ANY (SELECT selitem FROM fromitems WHERE whereitems)

can be transformed to the logically (but not necessarily semantically) equivalent form

< (SELECT MAX(selitem) FROM fromitems WHERE whereitems)

The same transformation is performed when the operator is <= or !>. Conversely,

< ALL (SELECT selitem FROM fromitems WHERE whereitems)

is transformed to the logically equivalent predicate

< (SELECT MIN(selitem) FROM fromitems WHERE whereitems)

and the same transformation is performed when the operator is <= or !>. If the operator is >, >=, or !<, the transformation is the reverse:

> ANY (SELECT selitem)

is transformed to

> (SELECT MIN(selitem))

and

> ALL (SELECT selitem)

is transformed to

> (SELECT MAX(selitem)).

More simply, a predicate of the form =ANY is transformed to IN, and a predicate of the form !=ANY is transformed to NOT IN.

9. Processing a General Nested Query

Algorithm NEST-JA2 applies to type-JA queries with a single level of nesting. The extension of the algorithm to type-JA queries with more than one level of nesting is not as simple as it was for algorithm NEST-N-J: the aggregate function and the join predicate may appear at any level of nesting, and not necessarily at the same level. Kim approaches the problem by means of query graphs: his algorithm NEST-G for transforming a general nested query gives the correct canonical result by inspecting and reducing the query graph for the query [KIM 82:465]. Rather than going into Kim's notations and methods, we will propose an alternative method for processing a general nested query, a direct postorder recursive algorithm which we believe is conceptually simple and which solves the problem of processing type-JA queries with greater than a single level of nesting.

9.1. Processing a General Nested Query: a Recursive Approach

The recursive version of algorithm NEST-G is described in the following pseudocode procedure nest_g(query_block), where the parameter query_block is a pointer to a SQL query block, possibly with descendant inner query blocks nested within it. The procedure is initially called with a pointer to the outermost query block (the beginning) of the query.
The following problem of correctly transforming a type-JA multiple levels of nesting. To demonstrate this, let us assume evaluates inner_query_block. replacing it with the transformed.

combining the recursion form by calling the appropriate transformation inner_query_block_type through the levels of a nested nodes useful to model a nested block beginning of the SQL statement) is the leaves'. Murill for each predicate in the WHERE clause of query_block

if predicate is a nested predicate (i.e. contains inner query block) nest_g(inner_query_block) /*
* Determine type of nesting, and call appropriate * transformation procedure.
*/
if SELECT clause of inner_query_block contains aggregate function if inner_query_block contains join predicate referencing a relation which is not in its FROM clause /*
* nesting is type-JA */
nest ja2(inner_query_block)
nest n_j(query_block,inner_query_block)
else /*
* nesting is type-A */
nest a(inner_query_block)
else
nest n_j(query_block,inner_query_block)
return

Three procedures are called by nest_g(): nest_a(), which evaluates inner_query_block, replacing it with the resulting constant; nest ja2(), which executes algorithm NEST-JA2; and nest n_j(), which executes Kim's algorithm NEST-N-J, combining the two query blocks query_block and inner_query_block. In explaining procedure nest_g() it is useful to model a nested query with a multi-way tree whose nodes are query blocks, where the outermost query block (the beginning of the SQL statement) is the root and the innermost query blocks are the leaves. Procedure nest_g() searches down through the levels of a nested query from the outermost query block until it finds the innermost query blocks (the leaves of the query tree). It then examines the leaf block to determine the type of nesting present, and transforms the parent to canonical form by calling the appropriate transformation procedures. After this is done for all nested predicates in query_block, the recursion then unwinds one level and the query block immediately above is processed in the same way, continuing the unwinding until lastly the outermost, or root, query block is transformed.

The algorithm represented in procedure nest_g() solves the problem of correctly transforming a type-JA query with multiple levels of nesting. To demonstrate this, let us assume the following query tree:

```plaintext
      (A)
     /  \
    A   N
   /   /  \
  J   J   N
 /    /   /  \
C    D    E
```

Figure 2: Example Query Tree

The nested iteration method of evaluating nested SQL queries can be inefficient for many queries: a relation referred to in an inner query block may have to be retrieved many times, possibly once for each tuple in the outer query block. Won Kim classified nested queries and proposed algorithms to reduce the cost of evaluating them [KIM 82]. The objective of his algorithms is to reduce the nested query to an equivalent single-level, or canonical, form. The resulting canonical query will contain explicit joins which capture the nested-iteration semantics of the original query, and can now be passed to a query optimizer which will determine an efficient order and method for the evaluation of the query. Kim compared the cost of evaluating a nested query by nested iteration and the cost of evaluating a transformed query using merge joins in several examples. The transformation method resulted in costs sometimes an order of magnitude smaller than the costs required by the nested iteration method. However, a bug in
Kim's algorithm NEST-JA was discovered by Werner Kiessling [KIE 84]. Another bug in the same algorithm has been demonstrated in section S. These bugs can be solved by performing a join in the creation of the temporary table which contains the aggregate information. If the aggregate function is COUNT, the join must be an outer join. This solution requires the join to be performed on a projection of the outer table in order to avoid an increase in the aggregate values due to duplicates in the outer table. The solutions to these bugs are incorporated into algorithm NEST-JA2, which retains Kim's strategy of building a temporary table to capture aggregate information, and which yields a cost reduction similar to that achieved by Kim in his example. The transformation algorithms have been extended to handle a larger class of predicates, and a recursive algorithm has been presented which will apply the transformations to a nested query of arbitrary complexity.

Acknowledgements

The authors would like to thank Professors Bruce J. McDonald and Marguerite C. Murphy for their helpful comments on an earlier version of this paper.

References


This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy. Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.