Improved Unnesting Algorithms for Join Aggregate SQL Queries

M. Muralikrishna

murali@cookie.enet.dec.com
Digital Equipment Corporation
CXO2-1, 301 Rockrimmon Blvd
Colorado Springs, CO 80919

Keywords: unnesting, optimization, SQL, the COUNT bug, outer join, anti-join, correlation predicate.

Abstract: The SQL language allows users to express queries that have nested subqueries in them. Optimization of nested queries has received considerable attention over the last few years [Kim82, Ganski87, Dayal87, Murali89]. As pointed out in [Ganski87], the solution presented in [Kim82] for JA type queries has the COUNT bug. In order to avoid the COUNT bug, the more general strategy described in [Ganski87] and [Dayal87] is used. In this paper, we modify Kim's algorithm so that it avoids the COUNT bug. The modified algorithm may be used when it is more efficient than the general strategy. In addition, we present a couple of enhancements that precompute aggregates and evaluate joins and outer joins in a top-down order. These enhancements eliminate cartesian products when certain correlation predicates are absent and enable us to employ Kim's method for more blocks. Finally, we incorporate the above improvements into a new unnesting algorithm.

1. Introduction

Traditionally, database systems have executed nested SQL [Astrahan75] queries using Tuple Iteration Semantics (TIS). It was analytically shown in [Kim82] that executing queries by TIS can be very inefficient. It was first pointed out in [Epstein79] and then in [Kim82] that nested queries can be evaluated very efficiently using relational algebra or set-oriented operators. The process of obtaining set-oriented operators to evaluate nested queries is known as **unnesting**.

It was later pointed out in [Kiessling84] and [Ganski87] that the unnesting techniques presented in [Kim82] do not always yield the correct results for nested queries that have non equi-join correlation predicates or for queries that have the COUNT aggregate between nested blocks. Unnesting solutions for these types of queries were provided in [Ganski87]. These solutions were further refined and extended in [Dayal87]. An important contribution of the current paper is a modification to Kim's algorithm that avoids the COUNT bug. Under certain conditions, Kim's approach may be more efficient than the general solution and hence worth considering.

In this paper, we focus our attention on unnesting Join-Aggregate (JA) type of SQL queries [Kim82]. These queries have correlation join predicates and an aggregate (AVG, SUM, MIN, MAX, or COUNT) between the nested blocks. The reason for focusing on JA type queries is that many other nesting predicates (such as EXISTS, NOT EXISTS, ALL, ANY) can be reduced to JA type queries [Ganski87, Dayal87]. An example of a 2 block JA type query is:

```
SELECT DEPT.name
FROM DEPT
WHERE DEPT.work_stations <
      (SELECT COUNT (EMP.*)
       FROM EMP
       WHERE DEPT.name = EMP.dept_name)
```

The above query finds the names of each of the departments that has more employees than work stations in it. The predicate 'DEPT.name = EMP.dept_name' is the correlation join predicate. Blocks may be nested within each other to any arbitrary depth. Notice that we

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

can associate a COUNT value with every tuple of DEPT. Section 3.2 of [Murali89] describes how the result of such a query can be obtained using TIS.

A few words on notation now. In the rest of this paper, upper case letters R, S, T, ... shall denote base relations. We refer to temporary relations as $TEMP_n$ where n is any positive integer. OP_n is used to denote any one of the six comparison operators ($\neq$, = , $\leq$, <, $\geq$, >). Attribute names are denoted by lower case identifiers, some of which may be just one letter long. We also assume that every base relation has a unique attribute denoted by the symbol #. Some unnesting algorithms require that every relation has at least one such attribute.

The reader is advised that we shall not adhere to strict SQL syntax when writing queries in this paper. The SQL syntax for expressing outer joins is fairly cumbersome. Instead, we shall write queries in a syntax that is fairly intuitive.

We introduce a couple of definitions here:

Definition: A **(Nested) Linear Query** is a JA type query in which at most one block is nested within any block.

Definition: A **(Nested) Tree Query** is a JA type query in which there is at least one block which has two or more blocks nested within it at the same level.

In this paper, we focus our attention on linear queries only. Extension to tree queries will be the topic of a future paper. The techniques for unnesting tree queries presented in [Murali89] were not as general as the ones we are developing in the current paper. For example, [Murali89] did not consider Kim's algorithm at all. For ease of notation, we shall assume that there is only one relation in the FROM clause of each block. The algorithms presented in this paper can be easily extended to the case when there are multiple relations in any FROM clause.

The rest of the paper is organized as follows: Section 2 provides background material briefly. In it we describe Kim's algorithm and show why it gives rise to the COUNT bug and how the solution provided in [Ganski87] eliminates the COUNT bug. We also describe the general solution presented in [Dayal87] that enables us to unnest linear queries. Section 3 discusses the enhancement to Kim's solution which avoids the COUNT bug and shows under what conditions an easy implementation is possible. We then describe, in Section 4, how the aggregate in the last block may be precomputed. Section 5 deals with evaluating joins and outer joins in the order they occur enabling us to employ Kim's method for a larger set of blocks. Finally, we incorporate the ideas presented in Sections 2 through 5 into a new integrated algorithm.

2. Related work

The purpose of this section is to familiarize the reader briefly with past work. For a thorough understanding of the earlier work, the reader is encouraged to read the following papers: [Kim82], [Ganski87], [Dayal87], and [Murali89].

2.1. Kim's algorithm and the COUNT bug

We motivate Kim's unnesting algorithm with the following example of a 2 block JA type query:

Example 1:
```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (S.*)
               FROM S
               WHERE R.c = S.c)
```

Kim's algorithm transforms the above query into the following two unnested queries.

Query 1:
```
TEMP1 (c, count) =
SELECT S.c, COUNT (S.*)
FROM S
GROUPBY S.c
```

Query 2:
```
SELECT R.a
FROM R, TEMP1
WHERE R.c = TEMP1.c AND R.b OP1 TEMP1.count
```

The result of the first query may be pipelined into the next query. Query 1 computes the COUNT value associated with every distinct value in the c attribute of S. Notice that a tuple of R may join with at most one tuple of $TEMP_1$. Kim's algorithm works correctly if the aggregate is not a COUNT. However, in the presence of the COUNT aggregate, the algorithm gives rise to the COUNT bug [Kiessling84, Ganski87]. A tuple r of R would be lost after the join if it does not join with any tuples of S. However, the COUNT associated with r is 0 and if (r.b OP_1 0) is true, r should appear in the result. In order to preserve tuples in R that have no joining tuples in S, an outer join[1] (OJ) is performed

when the COUNT aggregate is present between two blocks [Ganski87]. In this case, the unnested query becomes:

Query 3:
SELECT R.a
FROM R, S
WHERE R.c = S.c --- OJ
GROUPBY R.#
HAVING R.b OP_1 COUNT (S.*)

The outer join preserves every tuple of R and hence the COUNT bug is avoided. If it can be determined at compile time that R.b can never equal 0, then we could still use Kim's method. Notice that the outer join precedes the groupby operation in Query 3. Ganski's solution is more general than Kim's solution as the former may be applied even in the presence of non equi-join correlation predicates[2] [Ganski87].

2.2. Dayal's solution

The solution in [Dayal87] generalizes Ganski's solution for queries with more than 2 blocks. A linear query with multiple blocks gives rise to a 'linear J/OJ expression' where each instance of an operator is either a join or an outer join. A general linear J/OJ expression would look like:

R J/OJ S J/OJ T J/OJ U J/OJ ...

Relation R is associated with the outermost block, relation S with the next inner block and so on. An outer join is required if there is a COUNT between the respective blocks. In all other cases (AVG, MAX, MIN, SUM), we need perform only a join. The joins and outer joins are evaluated using the appropriate predicates.

Since joins and outer joins do not commute with each other in general[3], a legal order may be obtained by computing all the joins first and then computing the outer joins in a left to right order (top to bottom if you like) [Dayal87]. For example, the expression R OJ S J T J U OJ V J W can be legally evaluated as ((R OJ (S J T J U)) OJ (V J W)). Since we may evaluate the joins in any order, we may choose the cheapest join order to join relations S, T, and U.

Consider the following three block linear query.

```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (S.*)
               FROM S
               WHERE R.c OP2 S.c
               AND S.d OP3 (SELECT COUNT (T.*)
                            FROM T
                            WHERE S.e OP4 T.e
                            AND R.f OP5 T.f))
```

The corresponding linear expression is R OJ S OJ T and hence a legal order is (R OJ S) OJ T. The result is obtained by executing the following two queries.

Query 4:
$TEMP_1$ (#, a, b, *) =
SELECT R.#, R.a, R.b, S.*
FROM R, S, T
WHERE (R OJ S) OJ T
GROUP BY R.#, S.#
HAVING S.d OP_3 COUNT(T.*)

Query 5:
SELECT $TEMP_1$.a
FROM $TEMP_1$
GROUP BY $TEMP_1$.#
HAVING $TEMP_1$.b OP_1 COUNT($TEMP_1$.*)

The outer join predicates are implicit in Query 4. The predicate for R OJ S is (R.c OP_2 S.c), while the predicate for the second outer join with T is (S.e OP_4 T.e and R.f OP_5 T.f). Tuples from Query 4 may be pipelined into Query 5. The subtleties involved in Query 4 are explained in detail in [Murali89]. We also describe them briefly in Section 5 of this paper. Notice that if the query has d blocks, the total number of joins and outer joins will be (d-1). These will be followed by (d-1) groupby-having operations.

3. Modifying Kim's algorithm

In this section, we describe how Kim's algorithm may be modified to avoid the COUNT bug. The

---

[1]In this paper, an outer join will always signify a left outer join.

[2]For Kim's method to apply, only equi-join correlation predicates of the form $f_1(R) = f_2(S)$, must be present. f_1 and f_2 are functions that reference only R and S respectively.

[3][Rose90] shows that joins and outer joins commute under specific conditions. [Dayal87] introduced the notion of G-joins or generalized joins to commute joins and outer joins. In future, we plan to utilize G-joins and the ideas in [Rose90] to commute joins and outer joins while unnesting. The algorithms in this paper do not use G-joins explicitly.

motivation in trying to modify Kim's approach is that it may be more efficient than Ganski's solution. We first study queries with two blocks and then study queries with three or more blocks.

3.1. Queries with two blocks

We return to Example 1 in Section 2.1. Query 1, that created the temporary relation $TEMP_1$, remains unchanged. However, Query 2 has to be modified. We know that the COUNT associated with a tuple of R that does not join with any tuple of S is 0. Thus, a tuple of $r \in R$ that does not join with any tuple of $TEMP_1$ will be a result tuple if (r.b OP_1 0) is true. For a tuple $r \in R$ that joins with a tuple of $TEMP_1$, r will be a result tuple if (r.b OP_1 $TEMP_1$.count) is true. The join operator in Query 2 is replaced by an outer join. In addition, different predicates are applied to the join (matching) tuples and the anti-join (non-matching) tuples to determine if they belong to the result. Notationally, we write this as shown below:

Query 6:
SELECT R.a
FROM R, $TEMP_1$
WHERE R.c = $TEMP_1$.c --- OJ
[R.b OP_1 $TEMP_1$.count : R.b OP_1 0]

The square brackets, in the last line of the above query, enclose the two predicates which are separated by a colon. The first predicate is applied to the joining tuples while the second tuple is applied to the anti-join tuples. There is currently no way of expressing the above query in SQL.

We now show that under certain circumstances, the modified Kim's method may be more efficient than Ganski's method. The heuristic argument is based on (1) the number of tuples that flow from each node in the query plans corresponding to the two methods and (2) the number of tuples that have to be processed at each groupby and outer join node. The query plans for the two methods are shown in Figures 1 and 2. The edges in Figures 1 and 2 are labeled by the number of tuples flowing through those edges. Both methods involve accessing relations R and S. Clearly $|TEMP_1| \leq |S|$ and $|R| \leq |R \text{ OJ } S|$. Assume that $|S| < |R|$. The number of tuples flowing from the groupby node to the outer join node in Kim's method is equal to $|TEMP_1|$. The number of tuples flowing from the outer join node to the groupby node in Ganski's method is equal to $|R \text{ OJ } S|$. Clearly $|TEMP_1| < |R \text{ OJ } S|$. The

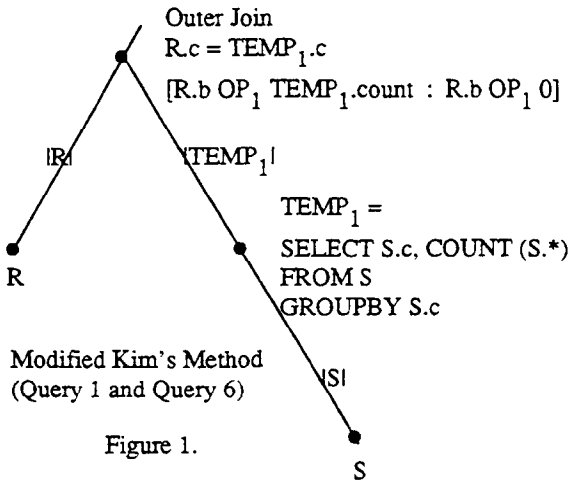

Figure 1.

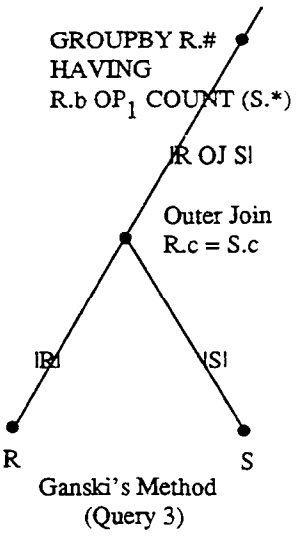

Figure 2.

number of tuples processed by the groupby node and the outer join node in Kim's method is each less than the corresponding number of tuples in Ganski's method. Hence if $|S| < |R|$, Kim's method should perform better than Ganski's method.

In the above discussion we have ignored the fact that Ganski's method joins two base relations, whereas in Kim's method, we join a base relation with a temporary relation. As a result, Ganski's method might be able to employ more join methods. Clearly, the optimizer has to pick the cheaper method more carefully than as outlined above. The important point is that we can use Kim's method even in the presence of the COUNT aggregate when the correlation predicates are all equi joins.

3.2. Queries with three blocks

We now extend the modified Kim's algorithm to queries with three blocks. We first introduce a definition here.

Definition: An equi-join correlation predicate is called a **neighbor predicate** if it references the relation in its own block and the relation from the immediately enclosing block.

Consider the following example in which all the join predicates are neighbor predicates.

Example 2:

```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (S.*)
               FROM S
               WHERE R.c = S.c
               AND S.d OP2 (SELECT COUNT (T.*)
                            FROM T
                            WHERE S.e = T.e))
```

The algorithm given in [Kim82] worked bottom up. We follow the same approach here. The result of the query is obtained by evaluating the following three unnested queries.

Query 7:

```
TEMP1 (e, count) =
SELECT T.e, COUNT (T.*)
FROM T
GROUPBY T.e
```

Query 8:

```
TEMP2 (c, count) =
SELECT S.c, COUNT (S.*)
FROM S, TEMP1
WHERE S.e = TEMP1.e --- OJ
    [S.d OP2 TEMP1.count : S.d OP2 0]
GROUPBY S.c
```

Query 9:

```
SELECT R.a
FROM R, TEMP2
WHERE R.c = TEMP2.c --- OJ
    [R.b OP1 TEMP2.count : R.b OP1 0]
```

Thus, we were able to extend the same principle to a three block query of Example 2 and avoid the COUNT bug. It is easy to see how we can extend the above solution to a query with more than three blocks as long as the correlation predicates are neighbor predicates. The natural question then is: what happens when we have non neighbor predicates. We address this in the next section.

3.3. Queries with non neighbor predicates

We start with the query shown in Example 3. This query is obtained by adding the non neighbor predicate, R.f = T.f, in the third block of the query in Example 2. Surprisingly, the query becomes very hard to unnest in the presence of the COUNT aggregates.

Example 3:

```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (S.*)
               FROM S
               WHERE R.c = S.c
               AND S.d OP2 (SELECT COUNT (T.*)
                            FROM T
                            WHERE S.e = T.e
                            AND R.f = T.f))
```

Evaluating bottom up, we would expect the three unnested queries to be as follows:

Query 10:

```
TEMP1 (e, f, count) =
SELECT T.e, T.f, COUNT (T.*)
FROM T
GROUPBY T.e, T.f
```

Query 11:

```
TEMP2 (c, f, count) =
SELECT S.c, TEMP1.f, COUNT (S.*)
FROM S, TEMP1
WHERE S.e = TEMP1.e  ---  OJ
    [S.d OP2 TEMP1.count : S.d OP2 0]
GROUPBY S.c, TEMP1.f
```

Query 12:

```
SELECT R.a
FROM R, TEMP2
WHERE (R.c = TEMP2.c AND R.f = TEMP2.f) --- OJ
    [R.b OP1 TEMP2.count : R.b OP1 0]
```

There are no surprises in Queries 10 and 12. In Query 12, each tuple of R joins with at most one tuple of $TEMP_2$. However, Query 11, as shown above, is incorrect! We shall return to this point soon. The objective of Query 11 is to compute COUNT (S.*) associated with every (c, f) pair. To better understand what Query 11 must really do, it is instructive to look at the

last two blocks of the query in Example 3 and evaluate COUNT (S.*) for various (c, f) values. Assume then that relations S and T are populated as shown below.

S	c	d	e
	10	2	100
	10	0	100
	10	1	200
	10	0	200
	10	3	200
	10	0	300

T	e	f	g
	100	1000	1
	100	1000	2
	200	1000	3
	200	2000	4

We only need to look at a single value in the c field as tuples with distinct c values fall into distinct groups. Using the above instances of S and T, we evaluate COUNT (S.*), from the last two blocks, for the following (c, f) groups as shown below:

c	f	COUNT (S.*)
10	1000	3
10	2000	3
10	3000	3

It is just a coincidence that the COUNT associated with every group is 3! Notice that the value 3000 is not present in the column T.f, but it may be present in R.f. We want Query 11 to produce such a table even though it has no knowledge of the R.f values.

We now return to discussing why Query 11, as shown, is incorrect. Notice that we are selecting attributes from both S and $TEMP_1$ in Query 11. We are also grouping by attributes from both the relations. In case an S tuple does not join with any $TEMP_1$ tuples, we cannot meaningfully evaluate the query. Let us try to understand what happens when an S tuple does not join with any tuple of $TEMP_1$. It is clear from the query of Example 3 that if an S tuple does not join with any T tuple, then COUNT (T.*) is 0, irrespective of the value of R.f. Therefore, such an S tuple will contribute to COUNT (S.*) if (S.d OP_2 0) is true.

There is another subtlety that we need to focus on. Assume that a tuple $s \in S$ joins with one or more $TEMP_1$ tuples. Let $\{TEMP_1.f\}$ denote the set of f values in the joining $TEMP_1$ tuples. We need to decide if s will contribute to COUNT (S.*). If a tuple $r \in R$ has as an f value that is in $\{TEMP_1.f\}$, we know that COUNT (T.*) associated with this (r, s) pair will be greater than 0. Then s will contribute to COUNT (S.*) if (s.d OP_2 $TEMP_1$.count) is true. On the other hand, for any tuple $r \in R$ that has an f value that is not in $\{TEMP_1.f\}$, the corresponding COUNT (T.*) will be 0. If (s.d OP_2 0) is true, then s will contribute to COUNT (S.*).

Using these observations, we now describe what the outer join operator of Query 11 must accomplish using the following pseudo code:

```
1  if no tuple of TEMP1 satisfies (s.e = TEMP1.e)
2  then output (s.c, all)
3  else for each tuple of TEMP1
4       satisfying (s.e = TEMP1.e)
5       {
6          if (s.d OP2 TEMP1.count)
7          then output (s.c, TEMP1.f)
8          else if (s.d OP2 0)
9               then output (s.c, ~{TEMP1.f})
10      }
```

The pseudo code focuses on one S tuple, s, at a time. The second component of the tuple in Line 2 is a set that denotes all possible values of f. In Line 7 we output a tuple of the form (s.c, $TEMP_1$.f). Line 9 indicates that we are outputting one tuple (s.c, $\sim\{TEMP_1.f\}$). The second component of the above tuple is a set of values and is equal to the complement of the values present in set $\{TEMP_1.f\}$. It is clear that the outer join operator has become more complex now!

The groupby operator in Query 11 is also a lot more complicated. The groupby operator can easily determine the group to which a tuple from Line 7 belongs to as both the c and f values are available. A tuple from Line 2 logically belongs to all groups that have the same c value as this tuple since its f value represents all possible values of f. A tuple from Line 9 belongs to all groups that have the same c value as this tuple but whose (the group's) f value does not belong to set $\{TEMP_1.f\}$. Logically, the number of such groups is bounded by the size of the domain of f. Potentially, this size could be infinite!

We further illustrate the complexity of the outer join and the groupby operations in Query 11 using the data stored in the relations S and T. We first use Query 10 to compute $TEMP_1$.

$TEMP_1$	e	f	count
	100	1000	2
	200	1000	1
	200	2000	1

Assuming OP_2 denotes equality, the outer join operator of Query 11 produces the following output.

c	second component	comments
10	1000	from the 1st tuple of S
10	~{1000}	from the 2nd tuple of S
10	1000	from the 3rd tuple of S
10	2000	from the 3rd tuple of S
10	~{1000, 2000}	from the 4rth tuple of S
10	all	from the 6th tuple of S

It should be easy to see that the first, third, and the last tuple in the above table belong to the (10, 1000) group. The second, the fourth, and the last tuple belong to the (10, 2000) group. Similarly, the second, the fifth, and the last tuple belong to the (10, x) group where x is any value except 1000 or 2000. The groupby operation of query 11 must take the output of the outer join operator and produce $TEMP_2$ (an infinite relation!) as shown below.

$TEMP_2$	c	f	count
	10	1000	3
	10	2000	3
	10	f_1	3
	10	f_2	3
	10	f_3	3
	...	...	...

None of the f_i's in the above table are equal to 1000 or 2000. Notice that in this example, for every c value we have generated all possible f values, and hence the predicate (R.f = $TEMP_2$.f) in Query 12 will always be satisfied. However, this predicate helps us identify the correct matching tuple in $TEMP_2$. We have not been able to develop an efficient implementation for the groupby operator of Query 11. Perhaps, it might be easier to modify the outer join of Query 12. Until a reasonable implementation is possible, we cannot employ Kim's method when a non neighbor predicate (T.f = R.f in this case) is present inside a COUNT block. However, if the second COUNT in Example 3 is replaced by a non COUNT aggregate, Query 11 would only have to perform a simple join. As in Dayal's solution, an outer join is used only when a COUNT aggregate is present between the blocks.

In the next two sections, we shall present a couple of strategies that will enable us to generate more plans. The goal we are working towards is a new unnesting algorithm in Section 6 that incorporates the ideas presented in Sections 2 through 5.

4. Precomputing the last aggregate

As we mentioned in Section 2.2, a valid J/OJ ordering is obtained by performing all the joins first, followed by the outer joins from left to right. Sometimes, we can change this order as demonstrated by the next query.

```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (S.*)
               FROM S
               AND S.d OP2 (SELECT MAX (T.d)
                            FROM T
                            WHERE R.f OP3 T.f)
```

The J/OJ expression for the above query is R OJ (S J T). Since there is no correlation predicate between the S and T blocks, we have to perform a cartesian product to compute (S J T). The outer join is then performed using the predicate (R.f OP_3 T.f). However, for each (r, s) pair, where r ∈ R and s ∈ S, MAX (T.d) depends only on r. Hence, we can precompute MAX (T.d) associated with each tuple of R as follows:

```
TEMP1 (#, a, b, max) =
SELECT R.#, R.a, R.b, MAX (T.d)
FROM R, T
WHERE R.f OP3 T.f --- OJ
GROUP BY R.#
```

Notice that | $TEMP_1$ | = | R |. Essentially, $TEMP_1$ has all the attributes of R required for further processing along with the MAX (T.d) associated with each tuple of R. We were able to compute MAX (T.d) in this fashion only because it occurred in the last block. Any aggregate that does not occur in the last block depends on the

results of the blocks below it and hence cannot be evaluated before the blocks below it are evaluated. Also, notice that we performed an outer join between R and T even though we were computing MAX (T.d). This is because COUNT (S.*) indirectly depends on each tuple of R as R is referenced inside the third block which is nested within the second block. Hence we must preserve all tuples of R. For a tuple of R with no joining tuples in T, the MAX value is set to NULL[4]. We can now rewrite the original query as follows:

SELECT $TEMP_1$.a
FROM $TEMP_1$
WHERE $TEMP_1$.b OP_1 (SELECT COUNT (S.*)
FROM S
WHERE S.d OP_2 $TEMP_1$.max)

We now have a correlation predicate between $TEMP_1$ and S, thus avoiding a cartesian product. The reader might note that similar ideas were presented in [Dayal87] in the section titled "Positioning G-Agg operations". In that section, [Dayal87] presents rules for computing aggregates before G-joins.

It is clear that it is possible to precompute the bottom most aggregate (BMA) if the number of outer relations referenced in the last block have already been joined. In the example of this section, the BMA depended only on one outer relation. In Section 6, we shall present an example where the BMA depends on more than one relation.

5. Performing outer joins before joins

As pointed out repeatedly, one correct evaluation order of a J/OJ expression is to perform the joins first followed by the outer joins from top to bottom. In this section we show that we may also proceed in a strictly top-down order, performing the joins and outer joins in the order they occur. As we shall see in Section 6, proceeding in a top down manner may enable us to use Kim's algorithm for a larger number of contiguous blocks at the end of the query. However, care must be taken to ensure that any join that is present just below an outer join is also evaluated as an outer join. We again illustrate with an example.

[4]Any comparison where one or both of the operands is NULL evaluates to unknown, which SQL regards as false for query evaluation purposes.

SELECT R.a
FROM R
WHERE R.b OP_1 (SELECT COUNT (S.*)
FROM S
WHERE R.c OP_2 S.c
AND S.d OP_3 (SELECT MAX (T.d)
FROM T
WHERE S.e OP_4 T.e
AND R.f OP_5 T.f))

The J/OJ expression is R OJ (S J T). The join predicate between S and T is (S.e OP_4 T.e) and the outer join predicate is (R.c OP_2 S.c AND R.f OP_5 T.f). Assume that the join between S and T is very expensive and should be possibly avoided. Could we evaluate (R OJ S) first? It turns out that we can indeed perform (R OJ S) first. However, some precautions/modifications are necessary.

It is clear that if an R tuple has no matching S tuples, the count associated with that R tuple is 0. As pointed out in [Murali89], this R tuple may be optionally routed to a higher node in the query tree so that it does not participate in the next join operation with T.

We thus need to consider only the join tuples of the form (r, s) from the outer join, where $r \in R$ and $s \in S$. Let us focus our attention on a single tuple r of R. When the join with T is evaluated using the predicate (S.e OP_4 T.e AND R.f OP_5 T.f), it is quite possible that none of these (r, s) tuples join with any tuples of T. In this case, the r tuple will be lost. However, if (r.b OP_1 0) is true, r is a result tuple and hence must be preserved. On the other hand, if some of the (r, s) tuples do join with some T tuples, it may so happen that after we do the groupby (as in Query 4 of Section 2.2) by (R.#, S.#) and evaluate MAX (T.d), none of the s.d values in the (r, s) tuples satisfy (s.d OP_3 MAX (T.d)). We may be tempted to discard all the (r, s) groups. Again if (r.b OP_1 0) is true, we need to preserve r.

We can preserve r if we perform the join between S and T as an outer join. Also, the groupby operator must not discard any (r, s) group not satisfying (s.d OP_3 MAX (T.d)). Instead, it must pass it on preserving the R portion of the tuple and nulling out the S portion of the tuple.

Similar ideas were used in [Murali89] when unnesting tree queries. Summarizing, if we encounter the expression R OJ S OJ T J U J V, we could evaluate it as ((R OJ S) OJ (T J U J V)). The above order

corresponds to evaluating all the joins first. Another evaluation order could be ((((R OJ S) OJ T) OJ (U J V)). Now we have an outer join between T and U. Carrying this idea one step further, the above expression may also be evaluated as ((((R OJ S) OJ T) OJ U) OJ V). As we shall see in the next section, joining relations in a top down order may enable us to employ Kim's method for a larger number of blocks.

6. An integrated algorithm

In this section we describe a new algorithm that generates execution plans by combining the ideas presented in Sections 2 through 5. We leave it to the optimizer to pick the cheapest plan. Before we describe the new algorithm, we introduce a fairly simple graphical notation for JA type queries. The new algorithm will operate on graphs.

The graph G = (V, E) for a JA type query consists of a set of vertices V and a set of directed edges E. There is a one-one correspondence between the blocks of the query and the elements of V. Each element of V, except for the first vertex, is labeled either C (COUNT) or NC (Non COUNT). This labeling is clearly suggestive of the kind of aggregate (COUNT or Non COUNT) present in that block. The vertices are numbered 1 through d, where d is the current number of vertices in the graph. A directed edge is drawn from vertex i to j (i < j) if there is a correlation predicate in the jth block between the relations of blocks i and j. In essence, the graph is a join graph.

Kim's method may be applied to the last k blocks of a query ($0 \leq k \leq d$) if the last k vertices of the graph of the query satisfy the following properties:

- The in degree of every C vertex is at most 1.
- The edge incident on a C vertex corresponds to a neighbor predicate.
- All the edges incident with the last k vertices correspond to equi-join correlation predicates.
- The relations in the first d-k blocks have already been joined.

The BMA may be precomputed if the in degree of the last vertex is at most 1.

The operations on the graph are as follows:

- When the relations of two or more blocks are joined, the corresponding vertices are collapsed into one vertex. The edges adjacent to these vertices are removed, while all the edges that connect these vertices to other vertices are preserved. Multiple edges are replaced by a single edge.
- Let d-1 and d be the last two vertices in the graph. If the BMA is computed, the last vertex d is removed from the graph and the edge incident on d is connected to d-1.

Notice that we may be able to apply Kim's method only after joining some relations. For example, we may apply Kim's method to the last block after joining R and S in the query of Example 3. This is because the predicate (R.f = T.f) becomes a neighbor predicate only after relations R and S are joined. Thus, the number of blocks for which we may apply Kim's method can change dynamically. Similarly, the BMA may have originally depended on more than one outer relation but after these relations have been joined, the in degree of the last vertex will become 1. The BMA may be precomputed at this point.

When a series of consecutive m joins are encountered in a J/OJ expression, one may be tempted to evaluate all the joins using the cheapest order. It will become evident from the example at the end of this paper that we must evaluate joins incrementally. In other words, we must evaluate the first i joins at a time, where $1 \leq i \leq m$. This ensures that we may be able to apply Kim's method to a larger group of contiguous blocks at the end of the query.

We are finally ready to present the new algorithm, **unnest**, in pseudo code. The input to the algorithm is the graph G of the query and the output is a set of query plans. We shall not describe how the output is specifically constructed as this is implicit in the operations on the graph and should be fairly self evident. References to G' in **unnest** denote the new graph derived from G.

```
unnest (G)
{
  if (the BMA can be precomputed)
     { compute the aggregate.
       unnest (G');
     }
   if (Kim's method can be applied to the
        remaining blocks)
     { apply Kim's method.
       return;
     }
   if (J---J---...---J---OJ---...) is encountered
     { for (i = 1; i <= m; i++)
          evaluate the first i joins using the
          cheapest join order.
          unnest (G');
     }
   if (OJ---J---J---...---J---OJ---...) is encountered
     { for (i = 1; i <= m; i++)
          evaluate the first i joins using the
          cheapest join order.
          unnest (G');

       evaluate the first OJ; replace the
       first J by OJ.
       unnest (G');
     }
}
```

We illustrate the working of the algorithm on the following query whose graph is shown in Figure 3.

```
SELECT R.a
FROM R
WHERE R.b OP1
        (SELECT COUNT (S.*)
         FROM S
         WHERE R.c = S.c
         AND S.d OP2
                 (SELECT AVG (T.d)
                  FROM T
                  WHERE S.e = T.e
                  AND R.f = T.f
                  AND T.g OP3
                          (SELECT SUM (U.g)
                           FROM U
                           WHERE S.h = U.h
                           AND T.i = U.i)))
```

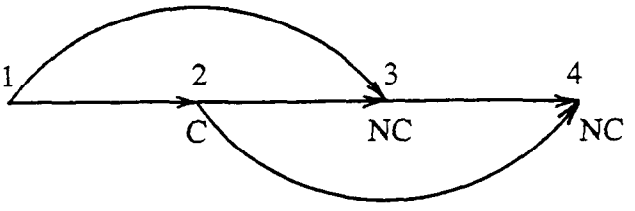

Figure 3.

The J/OJ expression is R---OJ---S---J---T---J---U. The following query plans, as shown in Figures 4(a)-4(f), are possible:

- (a) Apply Kim's method to blocks 2, 3, and 4.
- (b) Join R and S and apply Kim's method to blocks 3, and 4. Since the outer join between R and S is performed before the join, the first join is now evaluated as an outer join.
- (c) Join R, S, and T and apply Kim's method to block 4. Notice that both joins are now replaced by outer joins.
- (d) All joins have been replaced by outer joins, followed by three groupby operations.
- (e) Join relations S, T, and U first, followed by the outer join. This amounts to applying the general solution for the entire query.
- (f) Join relations S, T, and U first. Since the BMA depends only on relations S and T, the BMA is computed before the outer join with R.

Notice that it was important to evaluate the joins incrementally. Most of the outer join nodes in Figure 4 have two output edges. The vertical edge represents the anti-join tuples, while the other edge represents the join tuples. Similarly, the groupby-having nodes have two output edges. The vertical edge represents the groups that did not satisfy that condition in the having clause. These groups have certain portions nulled out. For example, in Figure 4(d), groups flowing from the first groupby-having node to the topmost groupby-having node along the vertical edge are of the form (R, NULL) [Murali89]. Also, Figures 4(b)-4(f) have edges that route tuples to a node much higher in the tree than the immediate parent. As pointed out in [Murali89], this is optional but leads to savings in message costs.

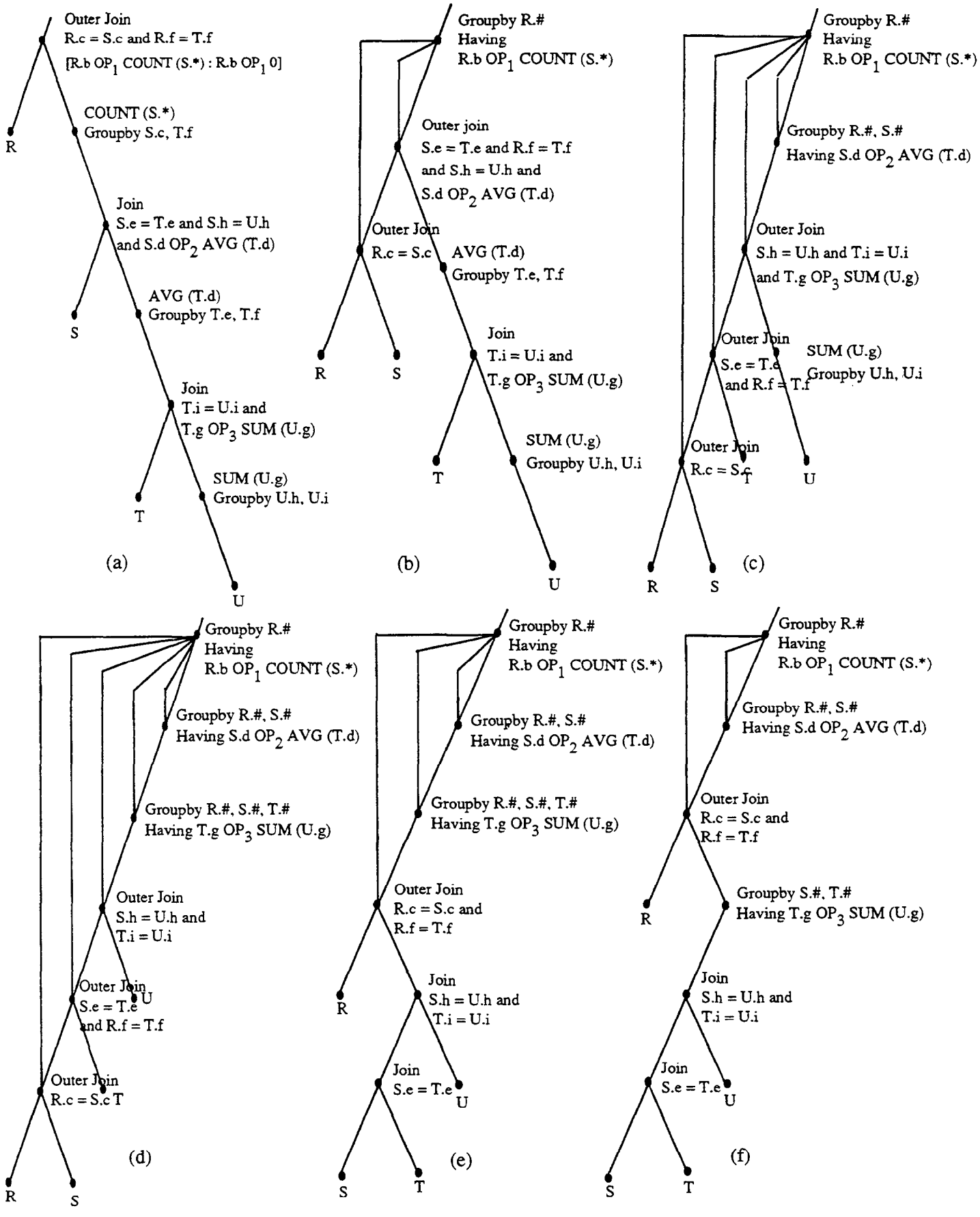
Outer Join
R.c = S.c and R.f = T.f
[R.b OP1 COUNT (S.*) : R.b OP1 0]
COUNT (S.*)
Groupby S.c, T.f
Join
S.e = T.e and S.h = U.h
and S.d OP2 AVG (T.d)
AVG (T.d)
Groupby T.e, T.f
Join
T.i = U.i and
T.g OP3 SUM (U.g)
SUM (U.g)
Groupby U.h, U.i
R
S
T
U
(a)
Groupby R.#
Having
R.b OP1 COUNT (S.*)
Outer join
S.e = T.e and R.f = T.f
and S.h = U.h and
S.d OP2 AVG (T.d)
Outer Join
R.c = S.c
AVG (T.d)
Groupby T.e, T.f
Join
T.i = U.i and
T.g OP3 SUM (U.g)
SUM (U.g)
Groupby U.h, U.i
R
S
T
U
(b)
Groupby R.#
Having
R.b OP1 COUNT (S.*)
Groupby R.#, S.#
Having S.d OP2 AVG (T.d)
Outer Join
S.h = U.h and T.i = U.i
and T.g OP3 SUM (U.g)
Outer Join
S.e = T.e
and R.f = T.f
SUM (U.g)
Groupby U.h, U.i
Outer Join
R.c = S.c
R
S
T
U
(c)
Groupby R.#
Having
R.b OP1 COUNT (S.*)
Groupby R.#, S.#
Having S.d OP2 AVG (T.d)
Groupby R.#, S.#, T.#
Having T.g OP3 SUM (U.g)
Outer Join
S.h = U.h and
T.i = U.i
Outer Join
S.e = T.e
and R.f = T.f
U
Outer Join
R.c = S.c
T
R
S
(d)
Groupby R.#
Having
R.b OP1 COUNT (S.*)
Groupby R.#, S.#
Having S.d OP2 AVG (T.d)
Groupby R.#, S.#, T.#
Having T.g OP3 SUM (U.g)
Outer Join
R.c = S.c and
R.f = T.f
Join
S.h = U.h and
T.i = U.i
Join
S.e = T.e
U
R
S
T
(e)
Groupby R.#
Having
R.b OP1 COUNT (S.*)
Groupby R.#, S.#
Having S.d OP2 AVG (T.d)
Outer Join
R.c = S.c and
R.f = T.f
Groupby S.#, T.#
Having T.g OP3 SUM (U.g)
Join
S.h = U.h and
T.i = U.i
Join
S.e = T.e
U
R
S
T
(f)

Figure 4.

7. Conclusions and future work

In this paper, we have presented a new algorithm that enhances and incorporates the previously known techniques for unnesting JA type queries. We are in the process of studying unnesting algorithms for all nested predicates in SQL. It appears that we can unnest the various SQL predicates using the techniques presented in this paper. It is hoped that more commercial systems will unnest SQL queries in the near future.

8. Acknowledgments

The author wishes to thank Umesh Dayal, Dan Dietterich, Christoph Freytag, Krishna Kulkarni, and Shirish Puranik for carefully reviewing earlier versions of this paper. The author is also grateful to the referees for their thoughtful comments. Their efforts have considerably improved the quality of the paper.

9. References

[**Astrahan75**] M. Astrahan, and D. Chamberlin, "Implementation of a structured English query language," Comm. of the ACM, Vol. 18, No. 10, (October 1975).

[**Dayal87**] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers," Proc. VLDB Conf., pp.197-202, (September 1987).

[**Epstein79**] Robert Epstein, "Techniques for processing of aggregates in relational database systems", ERL/UCB Memo M79/8, Electronics Research Laboratory, Univ. of California, Berkeley, (February 1979).

[**Ganski87**] Richard A. Ganski and Harry K. T. Long, "Optimization of nested SQL Queries Revisited", Proc. SIGMOD Conf., pp. 23-33, (May 1987).

[**Kiessling84**] W. Kiessling, "SQL-like and Quel-like correlation queries with aggregates revisited", UCB/ERL Memo 84/75, Univ. of California at Berkeley, (Sept. 1984).

[**Kim82**] W. Kim, "On Optimizing an SQL-like Nested Query", Trans. on Database Systems, Vol 9, No. 3, (1982).

[**Murali89**] M. Muralikrishna, "Optimization and Dataflow Algorithms for Nested Tree Queries", Proc. VLDB Conf., pp.77-85, (August 1989).

[**Rose90**] Arnon Rosenthal and Cesar Galindo-Legaria, "Query Graphs, Implementing Tress, and Freely-Reorderable Outerjoins", Proc. SIGMOD Conf., "pp. 291-299, (May 1990).