Optimization and Dataflow Algorithms for Nested Tree Queries

M. Muralikrishna
murali@cookie.dec.com
Digital Equipment Corporation
1175 Chapel Hills Drive
Colorado Springs, CO 80920

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

1. 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. Most of the previous optimization work has assumed that at most one block is nested within any given block. The solutions presented in the literature for the general case (where an arbitrary number of blocks are nested within a block) have either been incorrect or have dealt with a restricted subset of queries. The two main contributions of this paper are: (1) optimization strategies for queries that have an arbitrary number of blocks nested within any given block, and (2) a new dataflow algorithm for the execution of nested queries, involving one or more outer joins, in a multi-processor environment such as the one found in GAMMA. The new algorithm cuts down on message and CPU costs over conventional dataflow algorithms.

2. 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 [Kim82] that nested queries can be evaluated very efficiently using relational algebra operators or set-oriented operations. The process of obtaining set-oriented operations 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 function between nested blocks. Unnesting solutions for these types of queries were provided in [Ganski87]. These solutions were further refined and extended in [Dayal87].

In this paper, we will focus our attention on unnesting Join-Aggregate (JA) [Kim82] type queries. These queries have correlation join predicates and an aggregate function (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 JA type query is:

SELECT R_1.a
FROM R_1
WHERE R_1.b =
(SELECT COUNT (R_2.b)
FROM R_2
WHERE R_1.c > R_2.c)

The predicate (R_1.c > R_2.c) is the correlation join predicate. We will explain the meaning of these types of queries in the next section.

We introduce a couple of definitions here:

Definition 1: A **(Nested) Linear Query** is a JA type query in which at most one block is nested within any block.

Definition 2: 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.

Proceedings of the Fifteenth International
Conference on Very Large Data Bases

Amsterdam, 1989

It is worth pointing out that the unnesting solution presented in [Ganski87] for a linear query with more than two blocks is incorrect (see Section 4). [Dayal87] does not discuss tree queries.

The rest of the paper is organized as follows: In Section 3, we introduce the notation that we will use for JA type queries and explain the meaning of these queries using TIS. In Section 4 we will briefly summarize the results presented in [Dayal87] that enable us to unnest nested linear queries. We will present our solution for tree queries in Section 5. Section 6 discusses a new dataflow algorithm for nested queries in a multi-processor environment (such as the one found in GAMMA [Dewitt86], [Gerber86]). The new dataflow algorithm reduces processing and message costs. Finally, we will present an algorithm that takes a tree query as input and generates an execution tree that can be routed using our new dataflow algorithm.

3. Interpreting JA Type Queries

The easiest way to understand a general nested query is by means of Tuple Iteration Semantics (TIS). TIS provide an algorithm, albeit inefficient, for obtaining the result of a nested query. It is instructive, however, to interpret a nested query using TIS. In order to explain the meaning of JA type queries, we first define a notation which we will use in this paper to represent such queries.

3.1. Notation

A JA type query may be represented as a tree. Each node in the tree corresponds to a SQL query block. Query blocks that are nested within a parent block are represented as child nodes of the node corresponding to the parent block. For ease of explanation, we shall assume that each block has one relation in its FROM clause. By definition, a node is also its own ancestor. A predicate clause in a given block may reference a relation associated with any ancestor block. Predicate clauses may either be selection or join predicates.

The relation associated with block (or node) i is represented by R_i ($i > 0$). Lower case letters (a, b, etc.) represent attribute names. A '*' is used to denote all the attributes of a relation. $R_i.\#$ is some unique key of R_i. r_i, r_i', r_i'' are each used to denote a tuple of relation R_i. OP_n ($n > 0$) is any one of the following operators ($=, \neq, <, \leq, >, \geq$). $F_i(R_j)$ represents a selection predicate in the ith block on R_j. To simplify the notation, we will assume that all join predicates are binary[1]. A join predicate in the ith block is then represented as $F_i(R_j, R_k)$, where $j, k > 0$ and $j \neq k$. If a predicate in the ith block does not reference R_i, then it is called an outer predicate. In this paper we will assume that there are no outer predicates in our queries. Outer predicates can be handled as shown in [Dayal87].

3.2. Interpretation using TIS

Consider the following linear JA type Query.

Example 1: A Two Block Linear Query

```
SELECT R1.a
FROM R1
WHERE F1(R1)
AND R1.b OP1
        (SELECT COUNT (R2.*)
        FROM R2
        WHERE F2(R2) AND F2(R2, R1))
```

$F_1(R_1)$ and $F_2(R_2)$ are selection predicates on R_1 and R_2 respectively, while $F_2(R_2, R_1)$ is a correlation join predicate between R_1 and R_2.

A run time system that would execute the above query using TIS would proceed as follows: A tuple r_1 from R_1 would be fetched. If $F_1(R_1)$ is false for r_1, tuple r_1 will not be present in the result. Assuming $F_1(R_1)$ is true, the values of the relevant attributes of r_1 would be substituted into predicates at deeper levels ($F_2(R_2, R_1)$). The two block query now becomes a single block query

```
SELECT COUNT (R2.*)
FROM R2
WHERE F2'(R2)
```

$F_2'(R_2)$ is a predicate on R_2 and is equivalent to $F_2(R_2)$ AND $F_2(R_2, R_1)$ after values of r_1's attributes have been substituted in $F_2(R_2, R_1)$.

Let the COUNT value returned by this block be C ($C >= 0$). C represents the number of tuples of R_2 that satisfy $F_2'(R_2)$. If ($r_1.b$ OP_1 C) is true, r_1 will be in the result. Notice that each tuple of R_1 can occur in the result at most once. Using TIS, the system executes a query on R_2 (the inner relation) for every tuple of R_1 (the outer relation) leading to a very inefficient execution

[1] n-ary join predicates can be easily incorporated into the solutions presented in this paper.

strategy [Kim82].

One can easily interpret SQL queries with multiple levels of nesting using TIS. For example, consider the three block linear query shown in Example 2.

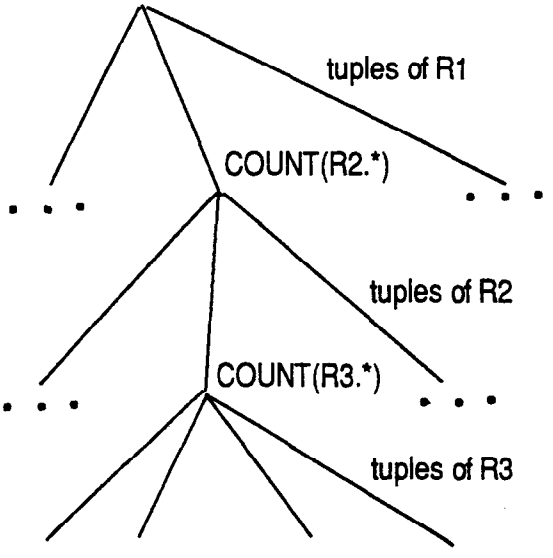
Example 2: A Three Block Linear Query

SELECT R_1.a
FROM R_1
WHERE $F_1(R_1)$
AND R_1.b OP_1
(SELECT COUNT (R_2.*)
FROM R_2
WHERE $F_2(R_2)$ AND $F_2(R_2, R_1)$
AND R_2.c OP_2
(SELECT (COUNT(R_3.*)
FROM R_3
WHERE $F_3(R_3)$ AND $F_3(R_3, R_2)$
AND $F_3(R_3, R_1)$)))

After reducing the above query to a two block query, one would substitute values from each tuple of R_1 and R_2 that satisfied $F_1(R_1)$ AND $F_2(R_2)$ AND $F_2(R_2, R_1)$ into the third block and evaluate COUNT(R_3.*). Those tuples of R_2 that satisfy R_2.c OP_2 COUNT(R_3.*) will be returned as part of the two block query. Notice that a COUNT(R_2.*) is associated with every relevant tuple of R_1 (satisfying $F_1(R_1)$) and a COUNT(R_3.*) is associated with every relevant pair of tuples of R_1 and R_2. This implies that the system would have to execute a query on R_3 for every relevant pair of tuples from R_1 and R_2. Pictorially this can be represented as shown in Figure 1.

The edges at level 1 in this tree represent all the tuples of R_1 that satisfy $F_1(R_1)$. Only those tuples of R_1 will be present in the result for which (R_1.b OP_1 COUNT(R_2.*)) is also true. Under each tuple of R_1, are those tuples of R_2 for which $F_2(R_2)$ AND $F_2(R_2, R_1)$ is true. Only those tuples of R_2 that also satisfy (R_2.c OP_2 COUNT(R_3.*)) will contribute to COUNT(R_2.*). Similarly, the tuples of R_3 under each relevant pair of tuples of R_1 and R_2 represent tuples that satisfy $F_3(R_3)$ AND $F_3(R_3, R_2)$ AND $F_3(R_3, R_1)$.

Figure 1: Pictorial Representation of the Three Block Query of Example 2



4. From TIS to Set-Oriented Semantics

In this section, we briefly summarize the general solution presented in [Dayal87] and show how it can be applied to unnest linear queries. For reasons of space constraints, we do not discuss the more specific solutions that are based on the strategies presented in [Kim82]. Besides, the solutions in [Kim82] are not general and hence can be applied only in special cases (as pointed out in Section 2). However, as pointed out in [Dayal87], the unnesting solutions presented in [Kim82] (when applicable) may yield a more efficient execution strategy than the general solution.

Let us now return to the Query of Example 1 in Section 3.2. A naive unnesting algorithm would join R_1 and R_2 using predicate $F_2(R_2, R_1)$ (after performing the respective selections first). The algorithm would then group the result by R_1.# (some unique key of R_1[2]) and compute COUNT(R_2.*) for each group and select only those groups associated with each tuple of R_1 that satisfy (R_1.b OP_1 COUNT(R_2.*)).

The naive algorithm gives rise to the COUNT bug [Kiessling84, Ganski87]. The COUNT bug arises if there are no joining tuples in R_2 for a particular tuple r_1 in R_1. r_1 would then be lost after the join. However, the COUNT associated with r_1 is 0 and if (r_1.b OP_1 0) is true, tuple r_1 should appear in the result. In order to

[2]A unique key is required in order to avoid the problem with duplicates in R_1 [Ganski87].

preserve tuples in R_1 that have no joining tuples in R_2, an outer join[3] (OJ) is performed when the COUNT function is present between two blocks [Ganski87].

A linear query with multiple blocks will give 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:

$$R_1 \text{ J/OJ } R_2 \text{ J/OJ } R_3 \text{ J/OJ } \ldots \text{ J/OJ } R_n$$

Relation R_1 is associated with the outermost block, relation R_2 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[4], 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]. Thus, the expression R_1 OJ R_2 J R_3 J R_4 OJ R_5 J R_6 can be legally evaluated as ((R_1 OJ (R_2 J R_3 J R_4)) OJ (R_5 J R_6)). Since we can evaluate joins in any order, we can choose the cheapest join order to join R_2, R_3, and R_4.

It is worth pointing out here that the solution presented in Section 9 of [Ganski87] for multiple level queries was incomplete in the sense that it does not discuss legal orderings when joins and outer joins are present in the same expression.

After all the joins and outer joins have been evaluated, the aggregate functions are evaluated in a bottom-up order after grouping the result by the appropriate unique keys. This is best illustrated with an example.

Consider the three block linear query of Example 2 in Section 3.2. The corresponding linear expression is R_1 OJ R_2 OJ R_3 and hence a legal order is (R_1 OJ R_2) OJ R_3. The predicate for R_1 OJ R_2 is $F_2(R_2, R_1)$ and the predicate for the outer join with R_3 is $F_3(R_3, R_2)$ AND $F_3(R_3, R_1)$.

We now show how the query of Example 2 can be evaluated using set-oriented operations. The result is obtained by executing more than one query. The result from one query may be pipelined to the next query. The two queries in this case are (not in strict SQL syntax!):

Query A: SELECT INTO TEMP
$R_1.\#, R_1.a, R_1.b, R_2.*$
FROM R_1, R_2, R_3
WHERE (R_1 OJ R_2) OJ R_3
GROUP BY $R_1.\#$, $R_2.\#$
HAVING $R_2.c$ OP_2 COUNT($R_3.*$)

Query B: SELECT $R_1.a$
FROM TEMP
GROUP BY $R_1.\#$
HAVING $R_1.b$ OP_1 COUNT($R_2.*$)

The results from Query A are fed into Query B. Even though the selection predicates ($F_i(R_i)$, i = 1, 2, 3) have not been shown in Query A, they are applied to the respective relations before they participate in the outer joins. The outer join predicates are also implicit in Query A.

4.1. A Few Subtleties

Query A has a few subtleties that were not mentioned in [Dayal87] and deserve to be highlighted. These subtleties will lead us to the development of the new dataflow algorithm (described in Section 6). The outer join between R_1 and R_2 results in two sets of tuples, viz., (R_1- X NULL)[5] and R_1R_2. R_1R_2 denotes the set $\{(r_1, r_2)$: $F_2(R_2)$ AND $F_2(R_2, R_1)$ AND $F_1(R_1)\}$, where the r_1 tuple $\in R_1$ and the r_2 tuple $\in R_2$. Let R_1+ denote the set of tuples of R_1 present in R_1R_2 (tuples of R_1 that participated in the join with R_2). R_1- denotes the set R_1 - (R_1+) (the tuples of R_1 in the anti-join).

Similarly, let $R_1R_2R_3$ denote the set $\{(r_1, r_2, r_3)$: $F_3(R_3)$ AND $F_3(R_3, R_2)$ AND $F_3(R_3, R_1)$ AND $F_2(R_2)$ AND $F_2(R_2, R_1)$ AND $F_1(R_1)\}$. Let the set of (r_1, r_2) tuples in R_1R_2 that joined with at least one tuple of R_3 be denoted by R_1R_2+. The set of (r_1, r_2) tuples that did not join with any tuple of R_3 is denoted by R_1R_2- and is

---

[3]When we talk about outer joins, we implicitly mean left outer joins.

[4]Dayal proposed the notion of generalized joins (G-Joins) to make joins and outer joins commutable but the equation given in the paper was incorrect. Without repeating the notation used in defining G-Join and the formal definition of G-Join, we simply state that the following equation was given in [Dayal87]: G-Join (R, G-Join(S, T; ∅; J2); R.*; J1) = G-Join(G-Join(R, S; R.*; J1), T; R.*; J2). However, it can be shown that this equation does not hold for the query in Figure 4.1 on page 202 in Dayal's paper.

[5]X represents the cartesian product operation.

equal to R_1R_2 - (R_1R_2+). Thus, the outer join with R_3 may yield up to three distinct sets of tuples, viz., (R_1- X NULL X NULL), (R_1R_2- X NULL), and $R_1R_2R_3$ respectively.

The (GROUP BY ... HAVING) operation in Query A has special semantics associated with it. For a given group of (r_1.#, r_2.#), if (r_2.c OP_2 COUNT(R_3.*)) is true, the (r_1.#, r_2.#) group is passed along to Query B. However, if (r_2.c OP_2 COUNT(R_3.*)) is false, the (r_1.#, r_2.#) group cannot be discarded. If the (r_1.#, r_2.#) is discarded and if this is the only group in which r_1 was present, COUNT(R_2.*) associated with the r_1 tuple is 0 and hence should be preserved. If (r_1.b OP_1 0) is true, r_1 will be part of the result. The (r_1, r_2) tuple that does not satisfy (r_2.c OP_2 COUNT(R_3.*)) should be passed along to Query B as (r_1, NULL). Similarly, for tuples in the set (R_1- X NULL X NULL), the GROUP BY ... HAVING operation passes them as (R_1- X NULL) to Query B because the predicate (r_2.c OP_2 COUNT(R_3.*)) is false as (NULL OP 0) is false.

5. Tree Expressions or Non Linear Expressions

So far we have restricted our discussion to linear queries only. If we permit more than one block to be nested within a given block at the same level (nested tree queries), we can get J/OJ expressions that are arbitrary trees. In this section, we will extend Dayal's solution to tree queries. A simple extension, albeit inefficient, to evaluate a tree expression would be the following: Choose an arbitrary path (perhaps the least expensive one) from the root to a leaf and evaluate the linear expression specified by this path as outlined in the previous section. This will yield a subset R_1' of the tuples of the root relation R_1. Using tuples in R_1', another path is evaluated yielding R_1'', a subset of R_1'. This is repeated until all paths are exhausted and the final set of result tuples are obtained.

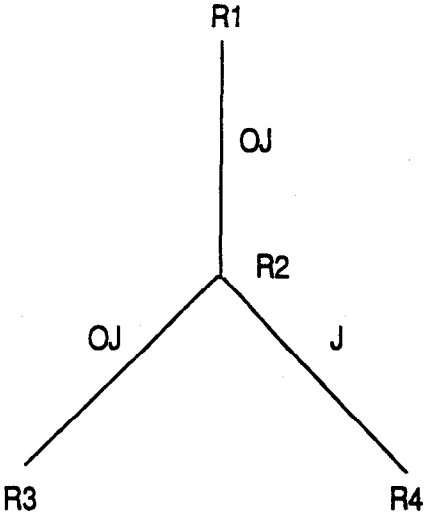
The above scheme is inefficient because relations that belong to two or more paths will be accessed more than once. For example, consider the tree query shown in Example 3 whose tree expression is shown in Figure 2. The edges in Figure 2 are labeled either by a J (denoting a join) or by an OJ (denoting an outer join).

Assume that the first path chosen is R_1--R_2--R_3. After evaluating the expression (R_1 OJ R_2) OJ R_3 and computing the respective aggregates bottom up, we will get a subset R_1' of tuples. Using these tuples, we take the other path. The J/OJ expression along this path is R_1' OJ (R_2 J R_4). Thus, the relation R_2 is accessed again. It would be ideal if each relation is accessed only once. After evaluating a J/OJ expression along one path, we would like to compute the aggregates bottom up only up to the point where a new branch begins. The idea is

Example 3: A Tree Query

```
SELECT R1.a
FROM R1
WHERE F1(R1)
AND R1.b OP1
        (SELECT COUNT (R2.*)
        FROM R2
        WHERE F2(R2) AND F2(R2, R1)
        AND R2.c OP2
                (SELECT (COUNT(R3.*)
                FROM R3
                WHERE F3(R3) AND F3(R3, R2)
                AND F3(R3, R1))
        AND R2.d OP3
                (SELECT (AVG(R4.d)
                FROM R4
                WHERE F4(R4) AND F4(R4, R2)
                AND F4(R4, R1)))
```

Figure 2: Tree Expression for the Query in Example 3



to use the tuples obtained thus far to evaluate the remainder of the J/OJ expression along the new path. However, this cannot be accomplished in a straight forward manner. Two kinds of anomalies may occur. We will illustrate these using the tree expression of Figure 2.

After evaluating (R_1 OJ R_2) OJ R_3 and the aggregate COUNT(R_3.*), there are two distinct sets of tuples. Tuples in the first set are of the form (r_1, NULL) and tuples in the second set are of the form (r_1, r_2) where $r_1 \in R_1$ and $r_2 \in R_2$.

Anomaly 1: Consider a tuple (r_1', NULL) from the first set. If r_1' is not present in the second set, COUNT(R_2.*) associated with r_1' is 0 and will be part of the result if (r_1'.b OP_1 0) is true. However, if (r_1', NULL) is joined with R_4 when the second path is taken, r_1' will be lost (because $F_4(R_4, R_2)$ is false since the R_2 fields are NULL).

Anomaly 2: Consider a tuple (r_1'', r_2') from the second set. If there is no tuple of R_4 such that $F_4(R_4, R_1)$ is true, then the r_1'' tuple will be lost after the join is computed. However, if the second path (R_1--R_2--R_4) was taken first, we would have got (r_1'', NULL, NULL) after evaluating R_1 OJ (R_2 J R_4). In this case, we do not lose r_1''. When the other path (R_1--R_2--R_3) is taken, r_1'' will not be lost as there is already an outer join between the R_2 and R_3 blocks.

The two anomalies demonstrate that tuples will be lost if a join is performed after evaluating an outer join. These tuples can be saved if the join between the R_2 and R_4 blocks is performed as an outer join. This leads to the following lemma.

Lemma: Let R_1--R_2--...--R_i--...--R_p and R_1--R_2--...--R_i-J-R_j--...--R_q be two paths from the root R_1 to the leaves R_p and R_q respectively in a tree expression. Let there be at least one outer join in the shared path R_1--R_2--...--R_i. Assuming we chose the R_1--R_2--...--R_p path first and there is a join between R_i and R_j, we can obtain the correct result by treating the join between R_i and R_j as an outer join.

Note that any joins below R_j are not affected as they will be evaluated before the outer join between the R_i and R_j blocks.

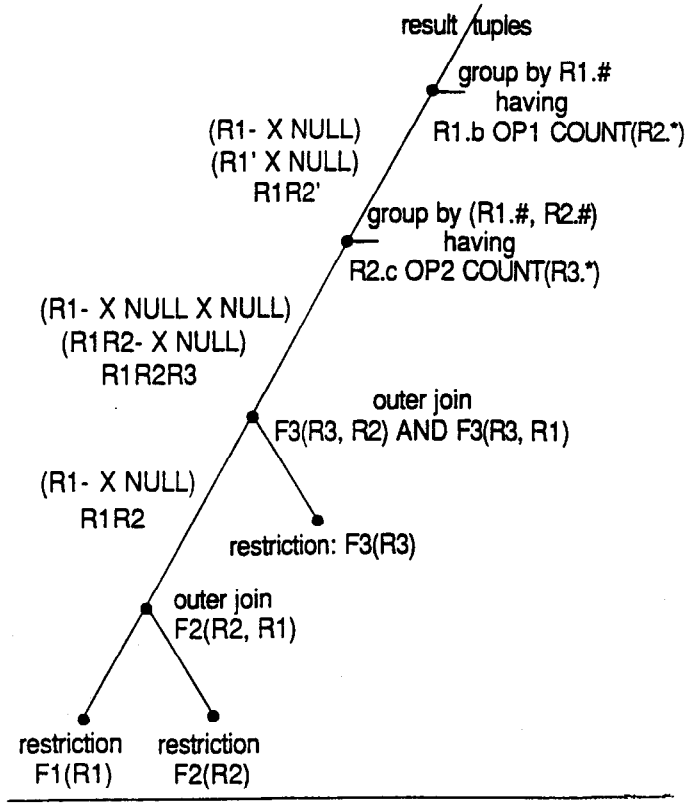
The proof for the lemma can be obtained by following the train of thought of the previous paragraphs and is omitted here. In summary, any join that comes after an outer join in a path must be evaluated as an outer join. Then each relation need be accessed only once.

6. An Improved Dataflow Algorithm

In this section, we will discuss a new dataflow algorithm for nested queries with COUNTs (and hence outer joins) in a multi-processor environment such as the one found in the GAMMA database machine [DeWitt86]. The new dataflow algorithm reduces message and processing costs over the conventional dataflow algorithm. The GAMMA database machine is a dataflow database machine that uses hash-partitioned algorithms for computing joins. A query with multiple joins can be routed very efficiently in GAMMA. The result of one join is pipelined to the next join operation. Details of the dataflow and scheduling algorithms of GAMMA can be found in [DeWitt86] and [Gerber86].

We will illustrate how GAMMA would have routed the three block query presented in Example 2, using the conventional dataflow algorithm. The

Figure 3: The Routing Method in GAMMA



optimizer in GAMMA sends the scheduler an optimized execution tree. The execution tree for the query in Example 2 is shown in Figure 3. The nodes in the execution tree represent operations (restriction/join/group by etc.), while the directed arcs represent information flow. Tuples always flow in an upward direction. In Figure 3, it is important to notice that the conventional dataflow algorithm sends the joining tuples as well as the anti-join tuples to the immediate higher (parent) node. The sets that are propagated between the operators of the execution tree of Figure 3 are shown along the respective edges (using the notation of Section 4.1).

The set R_1R_2' is derived from those tuples in $R_1R_2R_3$ and (R_1R_2- X NULL) that satisfy the predicate (R_2.c OP_2 COUNT(R_3.*)), while the set (R_1' X NULL) is derived from those tuples in $R_1R_2R_3$ and (R_1R_2- X NULL) that don't satisfy the above predicate. Notice that the attributes of R_2 have been replaced by NULL for these tuples.

In the presence of outer joins, a better execution tree may be obtained by sending the anti-join tuples to a node possibly higher than the parent node in the execution tree. This will result in savings in message and processing (CPU) costs. The new execution tree is shown in Figure 4.

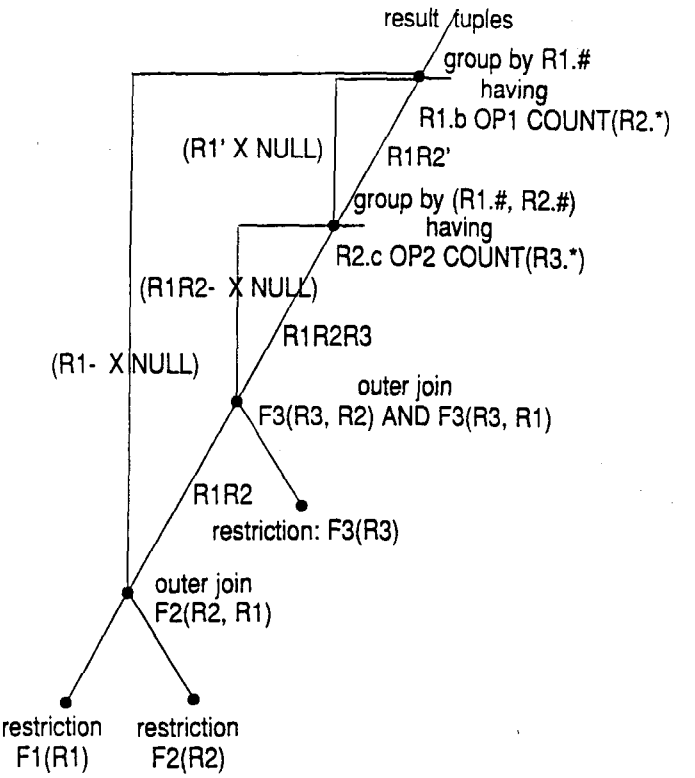
In the first execution tree (Figure 3), we are shipping the (R_1- X NULL X NULL) tuples (from the second outer join node to the first group by node) and the (R_1- X NULL) tuples (from the first group by node to the second group by node) unnecessarily. By doing so, we also incur the cost of processing them. In the second execution tree (Figure 4), the (R_1- X NULL) tuples are shipped directly from the first outer join node to the second group by node.

As the depth of nesting increases, the savings in message and processing costs will increase if the anti-join tuples are sent to a (possibly) higher node than the parent node. In the next section we will use the idea presented in Section 4.1 to send tuples that do not satisfy an aggregate predicate to a possibly higher node.

6.1. From J/OJ Tree Expressions to Execution Trees

We now describe how an execution tree is derived from a J/OJ expression tree. There are four different

Figure 4: The Improved Routing Method



kinds of nodes in an execution tree, each representing a different kind of operation. They are:

1. Restriction nodes (R nodes): These are leaf nodes in the execution tree and represent the restriction operation on the base relations.

2. Group By ... Aggregate nodes (GBA nodes): These nodes group the input tuples by the unique keys of the relevant relations and compute the aggregate for each group. Tuples that satisfy the aggregate predicate are sent to the immediate higher node and, if needed, tuples that do not satisfy the aggregate predicate are sent only to a GBA node that is possibly much higher in the execution tree (after nulling appropriate fields).

3. Join nodes (J nodes): The output of a join is fed to the immediate higher node in the execution tree.

4. Outer join nodes (OJ nodes): The joining tuples are sent to the immediate higher node, but the anti-join tuples (if needed) are sent only to a GBA node that is possibly much higher in the execution tree.

The outdegree of R and J nodes is one while the outdegree of remaining kinds of nodes is at least one and

at most two. The indegree of R nodes is 0, while the indegree of J and OJ nodes is two. The indegree of the GBA nodes is at least one. A GBA node can receive input from many internal nodes in the execution tree.

The number of leaf nodes in the execution tree is equal to the number of base relations in the query (or blocks in the query assuming that each block has exactly one relation associated with it). The number of internal nodes in the execution tree is equal to twice the number of edges in the J/OJ expression tree. Half of these nodes are J or OJ nodes while the other half are GBA nodes. This is because each edge in the J/OJ expression tree can be associated with a join/outer join and an aggregate operation.

Having determined the nodes in the execution tree, we need to define how the arcs are constructed. Each arc can be determined by its end nodes. The key is to also classify the nodes in terms of the structure of input tuples they expect and output tuples they produce. A directed arc will then be present between a node that produces a certain kind of output tuples and a node that expects that same kind of tuples as input.

6.2. Input/Output Classification of Execution Tree Nodes

1. R nodes: These leaf nodes only produce output. The output tuples from these nodes are qualifying single relation tuples. Attributes (from output at all nodes) not required for future operations are projected out.

2. J nodes: If relations R_i and R_j are joined at a J node, R_i and R_j tuples form the input. Output tuples have the form (r_i, r_j) where $r_i \in R_i$ and $r_j \in R_j$. R_i and R_j could be base relations or composite relations.

3. OJ nodes: If relations R_i (composed of relations $R_1, R_2, ..., R_n$) and R_j are outer joined at an OJ node, R_i and R_j tuples form the input. Joining output tuples have the form (r_i, r_j) where $r_i \in R_i$ ($r_i = (r_1, r_2, ..., r_n)$) and $r_j \in R_j$. One would expect the anti-join tuples to have the form $(r_i, NULL)$. This need not always be the case.

Consider the case when a J node is transformed to an OJ node. The anti-join tuple $(r_1, r_2, ..., r_n, NULL)$ can be discarded if there is no outer join in the path from R_1 to R_n. However, if the last outer join in the path from R_1 to R_n occurred between R_k and R_{k+1}, $0 < k < n$, the anti-join tuple $(r_1, r_2, ..., r_k, r_{k+1}, ..., r_n, NULL)$ is sent as $(r_1, r_2, ..., r_k, NULL)$ to the GBA node computing the COUNT of tuples of R_{k+1}. The COUNT(R_{k+1}.*) associated with the group $(r_1, r_2, ..., r_k)$ may be 0 and hence must be preserved.

We could have sent the tuple $(r_1, r_2, ..., r_n, NULL)$ to a much lower GBA node in the execution tree. However, since the aggregate at this GBA node is not a COUNT, the $(r_1, r_2, ...r_k, ..., r_n)$ tuple would not satisfy the aggregate predicate because of the presence of NULL. Hence the tuple will be passed to the next higher node as $(r_1, r_2, ..., r_k, ... r_{n-1}, NULL)$. However, we need only to preserve $(r_1, r_2, ..., r_k)$ and hence we can send $(r_1, r_2, ..., r_k, NULL)$ tuple directly to the corresponding GBA node.

4. GBA nodes: Assume that this GBA operation occurs between R_{n-1} and R_n in the path from R_1 to R_n. If the aggregate operation at this node was a non COUNT function, then the input to this node is of the form $(r_1, r_2, ..., r_{n-1}, r_n)$. If the aggregate function was a COUNT function tuples of the form $(r_1, r_2, ..., r_{n-1}, NULL)$ will also be included in the input. The output formats of a GBA node are exactly similar to those of OJ nodes.

6.3. An Example

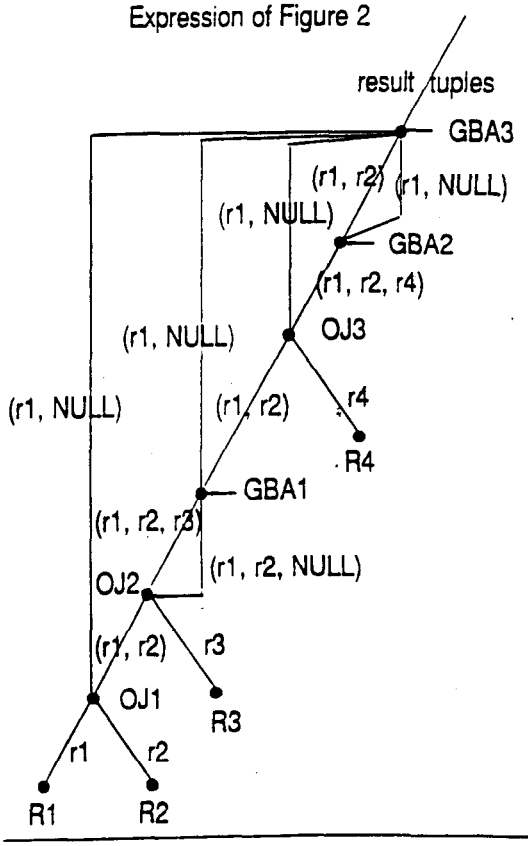
We will illustrate the above concepts in deriving an execution tree for the J/OJ expression in Figure 2. The execution tree will have four leaf nodes (R nodes) in the execution tree corresponding to the four base relations. There will be three pairs of J/OJ nodes and GBA nodes corresponding to the three edges. Assuming that we again take the left path (R_1--R_2--R_3) in Figure 2 first, the execution tree will be as shown in Figure 5. The nodes are labeled for ease of reference. Each arc is labeled with the format of tuples flowing along that arc. Notice that the 'anti-join' tuples of node OJ_3 are of the form $(r_1, NULL)$ rather than of the form $(r_1, r_2, NULL)$. As per the Lemma, the join at this node was converted to an outer join. It would not be useful to send tuples of the form $(r_1, r_2, NULL)$ to node GBA_2. Node GBA_2 would simply pass on these tuples to node GBA_3 as $(r_1, NULL)$. Instead, one saves processing and message costs by sending $(r_1, NULL)$ tuples directly from node OJ_3 to node GBA_3. It turns out that the operation at node OJ_3 is a G-Join [Dayal87]. Table 1 summarizes the savings realized in message costs.

Table 1.

Description	Size (in Bytes)	Conventional Routing Path	New Routing Path	Savings in Message Costs (in Bytes)
anti-join tuples from node OJ_1	M_1	OJ_1-OJ_2-GBA_1-OJ_3-GBA_2-GBA_3	OJ_1-GBA_3	$4*M_1$
tuples not satisfying aggregate comparison at node GBA_1	M_2	GBA_1-OJ_3-GBA_2-GBA_3	GBA_1-GBA_3	$2*M_2$
anti-join tuples from node OJ_3	M_3	OJ_3-GBA_2-GBA_3	OJ_3-GBA_3	M_3

Table 1 shows that for this example, the new dataflow algorithm results in a savings of ($4*M_1 + 2*M_2 + M_3$) bytes in message costs over the conventional dataflow algorithm. Further, a proportional savings is realized in processing costs.

Figure 5: Execution Tree for J/OJ Expression of Figure 2



7. Conclusions and Future Work

In this paper, we have presented a scheme for evaluating nested tree queries. We also described a new dataflow algorithm for such queries.

We are in the process of developing an efficient scheduling algorithm to go with our new dataflow algorithm. Our scheduling algorithm will be more complex than that of GAMMA's as we may to have to activate many operators in the execution tree simultaneously.

In this paper, we have assumed that blocks nested within a block at the same level are separated only by AND's. We are investigating optimization techniques for queries that have OR's between blocks that are nested at the same level.

8. Acknowledgments

The author wishes to thank Bob Gerber, Goetz Graefe, Krishna Kulkarni, Shirish Puranik, Jim Reuter, Mateen Siddiqui, Lynn Still, and Wai-Sze Tam for carefully reviewing earlier versions of this paper. 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).

[DeWitt86] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, "GAMMA: A High Performance Dataflow Database Machine," Proc. VLDB Conf., pp.228-237, (August 1986).

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

[Gerber86] R. Gerber, "Dataflow Query Processing Using Multiprocessor Hash-Partitioned Algorithms," PhD Thesis, Univ. of Wisconsin at Madison, (October 1986).

[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).