Reducing Multidatabase Query Response Time By Tree Balancing

Weimin Du, Ming-Chien Shan and Umeshwar Dayal
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304, USA
{du, shan, dayal}@hpl.hp.com

Abstract

Execution of multidatabase queries differs from that of traditional queries in that sort merge and hash joins are more often favored, as nested loop join requires repeated accesses to external data sources. As a consequence, left deep join trees obtained by traditional (e.g., System-R style) optimizers for multidatabase queries are often suboptimal, with respect to response time, due to the long delay for a sort merge (or hash) join node to produce its last result after the subordinate join node did. In this paper, we present an optimization strategy that first produces an optimal left deep join tree and then reduces the response time using simple tree transformations. This strategy has the advantages of guaranteed minimum total resource usage, improved response time, and low optimization overhead. We describe a class of basic transformations that is the cornerstone of our approach. Then we present algorithms that effectively apply basic transformations to balance a left deep join tree, and discuss how the technique can be incorporated into existing query optimizers.

1 Introduction

A multidatabase system (MDBS) integrates autonomous and possibly heterogeneous external data sources (EDSs). First, external database schemas are imported and integrated into one or more global views. Possible data and structural inconsistencies are resolved [DH84] [SL90] [SAD+94]. After the integration, users can issue queries against the integrated global views[1]. The MDBS decomposes a global query into subqueries against external database schemas and executes them.

As in traditional homogeneous distributed DBMSs (e.g., R* [LMH+85]), query optimization is important in MDBSs. Unfortunately, this problem has so far received little attention. Since the basic query optimization paradigm is the same in both environments, it is assumed that the traditional query optimization strategy (e.g., System-R style [SAC+79]) also works in MDBSs. While this is true to some extent, differences between MDBSs and homogeneous distributed DBMSs are significant enough to force us to re-examine certain query optimization issues in MDBSs. In this paper, we study the problem and propose techniques that extend the traditional optimization strategy to work for multidatabase queries.

1.1 The Problem

From a query processing point of view, the major difference between an MDBS and a homogeneous distributed DBMS is that EDSs in an MDBS are independent and autonomous DBMSs and were not designed to participate and cooperate in distributed query processing. This difference has great impact on multidatabase query execution.

For example, sort merge and hash joins are more often favored to implement global (inter-EDS) joins in MDBSs. Since each EDS is an independent and autonomous DBMS, the MDBS can only interact with it via its Application Programming Interface (e.g., at the SQL level). In other words, an MDBS is unable to access internal data structures and functions of its EDSs. Clearly, there is an overhead associated with each EDS access. Although the overhead is not signif-

[1] For this paper, we assume that the integrated views are relational.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery.To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGMOD ' 95,San Jose , CA USA
© 1995 ACM 0-89791-731-6/95/0005..$3.50

icant in a single access, it dominates the cost of operations that require repeated EDS accesses [DSD94A]. As a consequence, nested loop join can become expensive, and cannot be expected to out-perform sort merge join and hash join unless the outer table is very small.

On the other hand, sort merge and hash joins provide good inter-operator parallelism in a balanced (e.g., bushy) query tree, but bad response time in an unbalanced (e.g., left deep) query tree, due to long hierarchical delay of sort merge and hash joins. The hierarchical delay of a join node is defined to be the time span between the moment its subordinate join nodes produce their last results and the moment it does so. In general, join nodes implemented using sort merge and hash join incur long hierarchical delay, due to the time needed to sort input data and build hash table. Unfortunately, most traditional query optimization strategies generate only left deep join trees, and therefore are unacceptable in MDBSs, especially with respect to response time.

In this paper, we focus on this aspect of the multidatabase query optimization problem, namely improving response time of unbalanced left deep join trees. Our strategy works in two steps: first, a traditional cost-based (e.g., System-R style) optimization algorithm is employed to obtain a left deep join execution plan which is optimal with respect to total cost; then, the plan is improved (with respect to response time) using the proposed transformations. We describe basic transformations and present algorithms that effectively apply the transformations to a left deep join tree.

Compared to the existing optimization strategies, the proposed approach has the following advantages. First, it always selects better (at least no worse) execution plans (with respect to response time) than a traditional System-R style optimizer. Second, unlike the randomized optimization approach (e.g., [IK90]), it guarantees that the final execution plan is globally optimal with respect to total cost. This is the case as the second phase starts with a total cost optimal execution plan and each transformation will improve response time without increase of total cost. Finally, the optimization complexity of the proposed approach is much smaller than that of exhaustively searching the space of bushy trees. As a matter of fact, the complexity is comparable to that of the traditional optimization algorithm, which searches only left deep join trees, since the cost of the first phase dominates that of the second phase. Another advantage of our strategy is that it can easily be incorporated into an existing query optimizer.

1.2 Related Work

The idea of two phase optimization was also adopted by Hong and Stonebraker in [Hong92] [HS93] for parallel query optimization. However, our work differs from theirs in the following aspects. First, the purpose of the work in [HS93] is to parallelize a given query tree (left deep or bushy) without changing the tree structure, while our work is to transform a left deep join tree into a more balanced bushy join tree. Second, the two-phase optimization strategy is proposed in [HS93] to cope primarily with the problem of unknown run-time parameters at compile time and therefore the second phase must be performed at run time. The problem we are dealing with is poor response time due to unbalanced left deep join trees which is completely addressed at compile time.

Other papers have also studied the optimization problem for multidatabase queries. [BTY84] proposed query processing algorithms for multidatabase queries that use semijoin and take advantage of data replication. [Daya83] and [Daya85] discussed how traditional operations (e.g., selection, join, etc.) can be optimized in conjunction with generalization operations such as outer union and outer join. [Chen90] proposed heuristic rules that convert outerjoins introduced by integration into one-sided outerjoins, or even regular joins. [DKS92] discussed how the traditional System-R style optimization technique can be applied to multidatabase systems to produce optimal left deep join trees for multidatabase queries. The emphasis of [DKS92] is on calibration of EDS cost models and estimation of external subquery costs.

The problem of improving response time addressed in this paper is orthogonal to the problems studied in these other papers, as our techniques can be used in conjunction with any of their techniques.

1.3 Outline

Section 2 describes the basic idea and general approach of improving response time for a left deep join tree using tree transformations. The next two sections then focus on defining a class of basic transformations, and devising algorithms for applying a sequence of basic transformations. We discuss briefly in Section 5 how the technique can be incorporated into existing query optimizers.

2 Overview

There are two major factors that contribute to the poor response time of left deep join trees: sequential execution of joins and long hierarchical delay of sort

merge and hash joins. Since long hierarchical delay is inherent in sort merge and hash joins, the response time can only be improved by balancing left deep join trees so that joins can be executed concurrently. Fortunately, tree balancing is possible as joins are associative. We first illustrate the idea using an example. We then outline the general approach of balancing a left deep join tree.

2.1 Example

The example is taken from a product quality control application that analyses relationships between products, customers and suppliers. In the application, a company has a set of business divisions. Each division makes a number of products. An order is placed by a customer. For every order, the name of the product and the amount are registered. Finally, a product consists of a set of parts. Each part is used by a product and provided by a supplier.

Product(pid, name, did)*[2]
Division(did, name, city)
Order(pid, cid, quantity, date)
Customer(cid, name, city)
Part(tid, name, pid, sid*)*
Supplier(sid, name, city)

We assume that the six relations are imported from six different EDSs. The following query finds all products that have been sold to customers in San Francisco and that use parts provided by suppliers in San Francisco, and returns the names of the products and the divisions that make them.

> **select** *Pd.name, D.name* **from** *Product Pd, Division D, Order O, Customer C, Part Pt, Supplier S* **where** *C.city='S.F.'* **and** *S.city='S.F.'* **and** *Pd.pid=D.did* **and** *Pd.pid =O.pid* **and** *O.cid=C.cid* **and** *Pd.pid=Pt.tid* **and** *Pt.sid=S.sid*;

Figure 1 shows a possible left deep join tree generated by a traditional query optimizer for the query. We assume that all joins are implemented using the sort merge join method. Figure 2 shows a more balanced bushy join tree for the same query. The latter is generally considered to be better, especially with respect to query response time, as joins (i.e., those between *Product* and *Division*, *Order* and *Customer*, and *Part* and *Supplier*) are performed concurrently.

We note that it is possible to improve response time without increasing total cost. As a matter of fact, both response time and total cost could be improved by tree balancing, even if the original left deep join tree is already optimal with respect to total cost. This is the case because bushy join trees were not considered by the original optimizer.

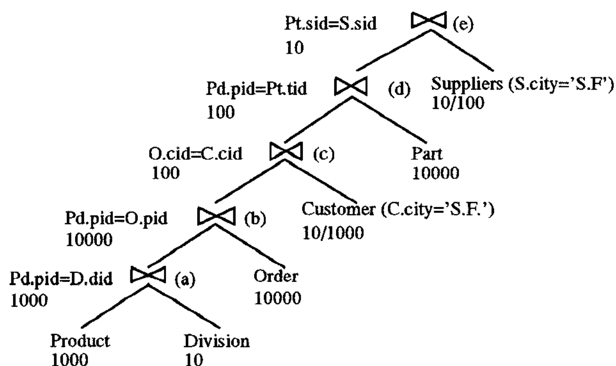

Figure 1: Left deep join tree

To see this, let us consider the join tree in Figure 1. Suppose that the company makes 1000 products, uses parts provided by 100 suppliers, and sells products to 1000 customers. For the sake of simplicity, we assume that each supplier provides 100 parts, each product consists of 10 parts, each customer places 10 orders, and each order consists of 10 products. Therefore, 1000 customers place a total of 10000 orders and 100 suppliers provide a total of 10000 parts. We also assume that 10 out of 100 suppliers and 10 out of 1000 customers are from San Francisco.

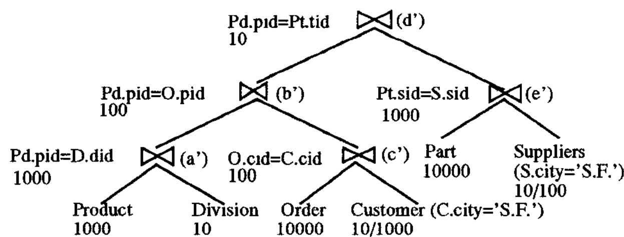

Figure 2: Balanced join tree

Joins (a') and (c') in Figure 2 have the same costs as joins (a) and (c) in Figure 1. However, the cost of join (b') is much smaller than that of join (b), as the size of join (c') is much smaller than that of table *Order*. Communication costs are also reduced in the balanced join tree. For example, to perform join (b) in Figure 1, we need to ship 1000 tuples (result of join (a)) to the site where *Order* resides. In Figure 2, however, we only need to ship 100 tuples (result of join (c')) to the site where the result of join (a) resides. Therefore, the total cost of evaluating the subtree rooted by join (c) in Figure 1 can be significantly reduced by transforming it into a more balanced bushy tree which allows the join between *Product* and *Division* and the join between *Order* and *Customer* to be performed concurrently.

[2]The primary key is underlined and the foreign keys are marked with asterisks.

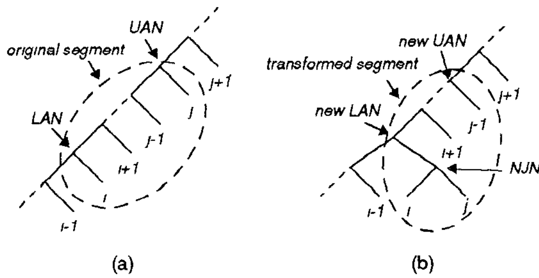

Figure 3: Basic transformation

We also note that not all transformations will improve query response time and total cost. In Figure 2, for example, the other transformation that allows join between *Part* and *Suppler* to be performed concurrently may increase total cost if the predicate on *Supplier* was (*S.city != 'S.F.'*). Join (e') between *Part* and *Supplier* in Figure 2 is expensive as *Part* is much bigger than join (d). Therefore, transformations cannot be blindly applied. Rather, they must be applied in a cost-based manner.

2.2 Balancing Left Deep Join Trees

Transforming a left deep join tree into a more balanced bushy join tree is not easy, due to the large number of possible bushy join trees. For a left deep join tree of n joins, there are $\frac{1}{n+1}\begin{pmatrix} n \\ 2n \end{pmatrix}$ equivalent bushy join trees. Clearly, we do not want to exhaustively search such a big transformation space. A feasible approach is to apply a sequence of *basic transformations* which can be easily identified and performed. In general, any complex transformation from a left deep join tree to an equivalent bushy join tree can be decomposed into a sequence of well defined basic transformations. For example, the transformation from the query tree in Figure 1 to the query tree in Figure 2 can be decomposed into two simpler ones. The first involves *Order* and *Customer*, and the other involves *Part* and *Supplier*.

For a given query tree, there are usually several possible transformations that may improve the response time. For a given transformation, there can also be many different ways of decomposing it into basic transformations. Different transformations may result in query trees of different response time and different decompositions may incur different overhead. As in the traditional optimization problem, the difference can be very large. Therefore, defining basic transformations and devising algorithms to control the application of basic transformations are two important issues in balancing left deep join trees using tree transformations. We discuss these two issues in the next two sections, respectively.

3 Basic Transformation

3.1 Definition

A basic transformation takes as input a tree segment (termed as *input segment*) from the join tree to be transformed. An input segment is identified by two anchor nodes termed as *upper anchor node* (UAN) and *lower anchor node* (LAN), respectively. The LAN of an input segment is a left descendant of the corresponding UAN[3]. The basic transformation replaces the input segment with the *transformed segment* which is similar to the input one except that (1) the left child node of the input UAN becomes the new UAN of the transformed segment, and (2) the LAN remains unchanged but its right child node is replaced with a new join node (NJN) of two subtrees which were the right child subtrees of the input UAN and LAN. Figure 3 illustrates the relationships between the input and transformed segments.

For example, the transformation from the left deep join tree in Figure 1 to the bushy join tree in Figure 2 consists of two basic transformations. The input segment of one basic transformation is identified by join (e) as the UAN and join (d) as the LAN, while the other is identified by join (c) as the UAN and join (b) as the LAN. In this particular example, the new UANs and their corresponding LANs happen to be the same for each transformed segment: join (d') for the former and join (b') for the latter.

For a given query tree, there are many possible applicable basic transformations. We are only interested in those that are both valid and cost improving. A basic transformation is *valid* if it preserves the semantics of the original query tree. In other words, the transformed query tree should evaluate the same query as the original one. A transformation is invalid if it causes a join node to have insufficient or improperly ordered input data. In Figure 1, for example, join (b) depends on tables *Product* and *Order*. Any transformation that moves *Product* or *Order* above the join node is invalid.

Clearly, a basic transformation does not cause insufficient input data to any joins, as it moves UAN's right child subtree down. Joins between UAN and LAN (e.g., joins $i+1$, ..., and $j-1$ in Figure 3) that may depend on input from LAN can still be properly computed. Unfortunately, a basic transforma-

[3] Left descendants of an internal node are recursively defined as follows. The left child node of a node is its left descendant. The left child node of any left descendant of a node is also a left descendant of the node.

tion may result in improperly ordered intermediate results. More specifically, the order of the NJN result may be different from that of LAN's original right child node, and the result order of the new UAN may be different from that of the input UAN. The former is possible as the LAN's right child subtree has been replaced with the NJN, and the latter is possible as the input UAN has been moved. Sort nodes may be needed in the two places to put the results into the original orders. With properly added sort nodes, the new UAN produces exactly the same data as the input UAN.

A basic transformation is *cost improving* if the response time of the transformed join tree is smaller than that of the original join tree and the total cost of the transformed join tree is not greater than that of the original join tree. A transformation may or may not be cost improving, a decision that can only be judged based on cost estimation. In cases where sort operations have been introduced as the result of transformation, their costs should also be included in the cost of the transformed join tree. We note that most of the cost information of the original join tree can be reused, except for the NJN and those join nodes between the UAN and LAN. Since the size of the segment is usually small (see the next subsection), the overhead should not be significant. The costs of the ancestor nodes of the UAN can be derived efficiently from the old ones by simply subtracting the cost difference between the original and the transformed UANs. This is possible as the result of the UAN remains unchanged in the transformation. Costs of all other nodes are unchanged in the transformation.

Basic transformations are desired for two major reasons: they are easy to implement and they improve query response time. There are two aspects of easy implementation. First, basic transformations are easy to identify. For a given UAN, the corresponding LAN is selected by examining its left descendants only. Second, they are easy to perform, as a basic transformation affects only a small portion of the join tree (i.e., join nodes between UAN and LAN). Since the transformation preserves the result of the input segment (with proper sort nodes), all other nodes are not affected, except for the execution costs of UAN's ancestors.

The primary use of a basic transformation is to balance left deep join trees. It does so by converting n sequentially executed joins (between and including the UAN and the LAN) into $n-1$ sequential joins (between and including the LAN and new UAN) and a new join concurrent to the left child subtree of the LAN. The assumption is that for a mostly left deep join tree, the right child subtrees of the UAN and the LAN are usually smaller than LAN's left child subtree. By combining UAN's right child subtree and LAN's right child subtree, the original join tree is balanced (especially for the LAN). The query response time may be improved as the number of sequentially executed joins between the UAN and the LAN has been decreased.

3.2 Identifying Basic Transformations

As we have seen, a basic transformation is uniquely identified by the UAN and the LAN. The process of balancing a left deep join tree is therefore a process of repeatedly selecting UANs and LANs and applying the basic transformations. In the previous subsection, we have seen how a basic transformation is applied to an unbalanced join tree. In this subsection, we discuss the issue of selecting a LAN for a given UAN. Selecting UANs is the subject of the next section.

There are two basic criteria in selecting LANs: the procedure should be simple, and the resulting NJN should be balanced. The two criteria, however, are somewhat contradictory. Simplicity implies selecting a LAN that is close to the UAN, while balance implies selecting a LAN whose right child node is closest (in terms of response time) to that of UAN's right child subtree. A practical solution is to select the closest node whose right child subtree's response time is within a certain range (relative to that of the UAN's right child subtree).

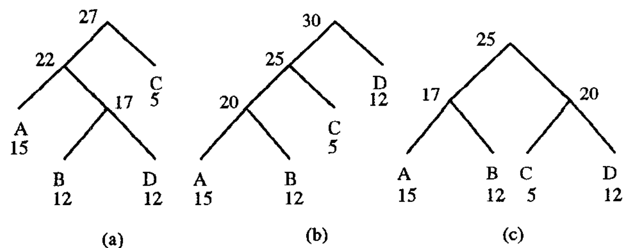

Figure 4: Selecting LAN

In the above described scheme, nodes under the UAN will be skipped if their response times are out of the pre-specified range. This could cause a problem as the following example shows. Figure 4.b shows a join tree of four tables: A, B, C, and D. The cost of each table and the response time of each join node are shown in the figure. We assume that each join takes 5 units of work. Suppose the root is the current UAN, Figure 4.a shows the result of the basic transformation as selected by the above algorithm with range 12 ± 5. Table C is not selected because its response time is out of the range. Suppose there is a join predicate between tables C and D, but no join predicate

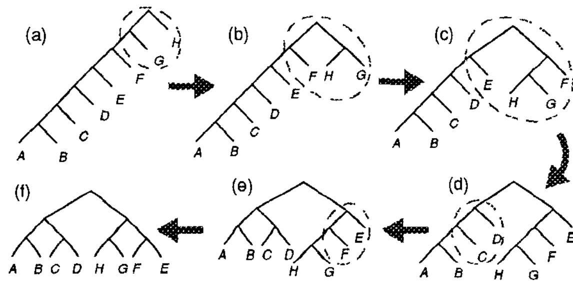

Figure 5: Example of top-down transformation

between tables A and C. Although the NJN subtree of the previous transformation is balanced, the overall join tree is not balanced and cannot be further balanced, as the cross product of tables A and C is very expensive. The problem can be fixed by extending the algorithm to skip only those join tables that have join predicates with other join tables (e.g., A). Figure 4.c shows a more balanced join tree produced by the basic transformation that is selected by the extended algorithm.

In the following discussion, we only consider the basic transformations that are both valid and cost improving. We will simply refer to them as basic transformations. We also assume the existence of the following two functions: Find-LAN(U) and Transform(U, L). Given a UAN U, the first function returns L as the LAN using the above algorithm (or **null** if no such node can be found), and the other applies the basic transformation identified by U and L to the subtree rooted by U. The left child node of U becomes the new UAN after the transformation.

4 Algorithms

In this section, we describe three tree traversing algorithms that govern the selection of UANs. The first algorithm (top-down approach) traverses a join tree from the root to the leaves, while the second (bottom-up approach) does it from the leaves to the root. Brief analyses follow showing the strengths and weaknesses of each approach. A hybrid approach is then proposed which overcomes these weaknesses.

4.1 Top-Down Approach

4.1.1 Algorithm

The intuition behind the top-down approach is that a join tree can be balanced by evenly distributing the load of each internal join node between its two child nodes.

In the algorithm, the root of the original left deep join tree serves as the initial UAN. For each UAN, there are three cases. The first case is characterized by a leaf left child node. The algorithm simply terminates as the load of the UAN cannot be redistributed and all other join nodes (which are ancestors of the current UAN) have already been processed.

The second case is when the response time of the right child subtree is about the same as or greater than that of the left child subtree. In other words, the load of the UAN has already been distributed between its two child subtrees as evenly as possible. Although no transformation is necessary for the UAN, the two child subtrees may still be unbalanced. Recursive invocations of the algorithm on both child subtrees are therefore performed.

A transformation is needed only in the last case when the response time of the left child subtree is greater than that of the right child subtree. If there exists such a basic transformation, it will be performed. UAN's left child node becomes the new UAN after the transformation. If no transformations could be found for the current UAN, the algorithm will first balance the left child subtree as it dominates the overall response time. The right child subtree will also be balanced if it dominates the overall response time after the left child subtree has been balanced.

4.1.2 Example

Figure 5 shows an example of tree balancing using the top-down approach, where the input segment (and therefore the UANs and LANs) are circled at each step. For the sake of simplicity, we assume that the eight join tables are about the same size. We further assume that there exists a join predicate between each pair of join tables. At the beginning, the left child node of the root is much larger than the right one. Therefore, the root is selected as the UAN and the parent of table G is selected as the LAN (Figure 5.a). The transformation has the "effect" of moving table G from left child subtree into the right child subtree (Figure 5.b). The process continues until both child subtrees are left deep join trees of four tables (Figure 5.d). Although the load of the new root is evenly distributed, the overall join tree is not balanced, as both its child subtrees are still left deep trees. Further balancing on each of the two child subtrees is performed (Figure 5.d and Figure 5.e) and the final join tree is shown in Figure 5.f.

4.1.3 Discussion

There are two major issues in balancing left deep join trees: the quality of the resulting bushy join trees and the overhead it incurs. The former can be measured by the improvement in the response time, while the

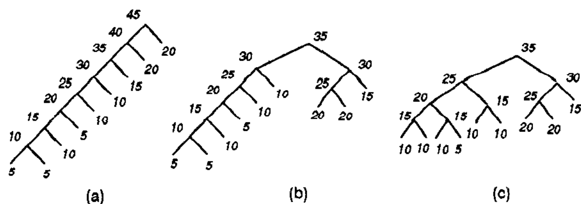

Figure 6: Problem with top-down transformation

latter can be measured by the number of basic transformations it invokes. We now discuss the two issues in the context of the top-down algorithm.

The overhead incurred by the top-down approach is minimal, especially compared to that of exhaustive search. It can be shown that for any given left deep join tree of n joins, the top-down approach performs at most $\frac{1}{2}(n-1)(n-2)$ basic transformations (see [DSD94B] for details).

In the general case where the loads of leaf nodes are similar, a left deep join tree can be transformed into a balanced bushy join tree in about $O(n \log n)$ basic transformations. This is the case as $\log n$ steps are needed, and at each step, n basic transformations are needed to balance the join nodes at the level.

Unfortunately, the top-down approach does not guarantee that the resulting join tree is optimal with respect to response time. It only improves the response time of a given left deep join tree. One problem with the top-down approach is that it balances a node before its child nodes are balanced. Therefore, a node is balanced based on the response times of its unbalanced child subtrees. However, two subtrees of the same response times may have different response times after being balanced. Figure 6 shows such an example, where Figure 6.a is the original left deep join tree. In the example, each table node is labeled with its cost and each join node is labeled with the response time of the subtree rooted by the node. We assume that each join requires 5 units of work (including sorting the join result). Figure 6.b shows the result of the first step of the transformation (i.e., the load of the root has been evenly distributed). In this example, the left child subtree can be further balanced, while the right child subtree cannot. Figure 6.c shows the final bushy join tree which is not well balanced. We will see a more balanced join tree in Subsection 4.3.3.

4.2 Bottom-Up Approach

4.2.1 Algorithm

In contrast to the top-down approach, the bottom-up approach balances UANs after their child subtrees have been balanced.

Since the right child nodes are initially leaf nodes in left deep join trees, the algorithm first recursively balances the left child subtree. The process stops when the UAN is the parent of the leftmost leaf node. The algorithm returns to the parent of the current UAN either immediately, if there exists no valid and cost improving basic transformation for the UAN, or after each transformation if there exists one. Therefore, the UAN moves from the parent of the leftmost leaf node to the root of the original join tree.

At each UAN, if a transformation can be identified to improve the overall query response time, it will be performed. The transformation has the effect of adding the right child subtree into the (already balanced) left child subtree, resulting in a bigger balanced subtree.

It is possible that the NJN is not balanced after the transformation, as it is formed by joining the right child subtrees of UAN and LAN. UAN's right child subtree is always a leaf node but LAN's right child subtree can be an arbitrarily complicated subtree. Further balancing on the NJN might be necessary. Of course, the balancing is necessary only if the NJN dominates the response time of its parent node.

If no valid and cost improving transformation can be found at the current UAN, the algorithm usually returns. There are, however, cases where transformations are desired even if they do not improve the overall query response time. The transformation can be justified if the LAN is not well balanced. The idea is that the overall response time could be improved later by further balancing the NJN. Note that the non-cost improving transformation is always followed by an attempt to further balance the NJN. This is the case as the NJN always dominates the cost of its parent node after each such transformation.

4.2.2 Example

Figure 7 shows the bottom-up transformation that balances the same left deep join tree as in Figure 5.a. The algorithm starts from the root of the original query tree. The UAN moves down from the root to the parent of nodes A and B, as the algorithm recursively invokes itself to balance the left child subtrees. Since both nodes A and B are leaf nodes, their parent node cannot be further balanced. The algorithm therefore returns and the parent of node C becomes the UAN. Although the UAN's left child subtree is more heavily loaded than the right child subtree, no basic transformation can be found to either improve the response time or further balance the LAN. The first cost improving basic transformation is

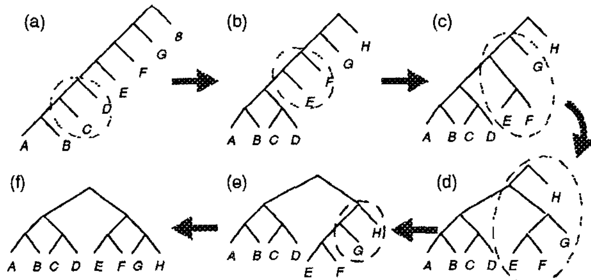

Figure 7: Example of bottom-up transformation

found when the parent of node D becomes the UAN (Figure 7.a). C's parent node and node D serve as the LAN and UAN, respectively, in the transformation. The transformation has the effect of adding the UAN's right child node (node D) into UAN's left child subtree, resulting in a new balanced subtree (containing nodes A, B, C and D). The parent of node E becomes the new UAN after the algorithm returns. Clearly, there exists no cost improving transformation. In addition, all possible LAN candidates are already perfectly balanced. Therefore, no transformation will be performed. The process continues until the root becomes the current UAN (Figure 7.d). Note that the transformation from Figure 7.d to Figure 7.e does not actually improve the overall query response time. The transformation is nevertheless performed, because the LAN is not balanced. In this particular example, the NJN of the transformation can be further balanced which results in a better join tree (Figure 7.f).

Please note that, in general, the top-down and bottom-up approaches result in different bushy trees. In the cases where they produce the same bushy join tree, they may still invoke different basic transformations, as in the previous examples.

4.2.3 Discussion

The bottom-up approach works by adding the right child subtree of the UAN into the (already balanced) left child subtree and always results in a bigger balanced subtree after the transformation. Since it starts from bottom, it lacks the global view of the join tree. It is therefore possible that a subtree is perfectly balanced but the overall query tree is not, as shown in Figure 8. The top-down approach is clearly superior in this case.

The upper bound complexity of the bottom-up approach is $\frac{1}{2}(n-1)(n-2)$, similar to that of the top-down approach, and the complexity for the general case is $O(n \log n)$ [DSD94B].

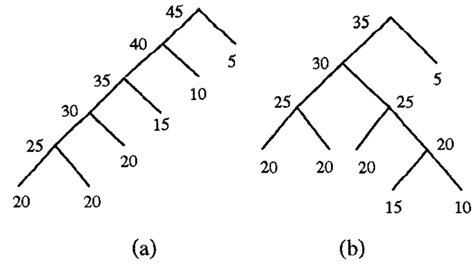

Figure 8: Problem with bottom-up transformation

4.3 Hybrid Approach

4.3.1 Algorithm

As we have seen, both the top-down and bottom-up approaches have their weaknesses. The following algorithm (Figure 9) outlines a hybrid approach that combines the top-down and bottom-up approaches to overcome these weaknesses. A left deep join tree with root N can be balanced by invoking Hybrid(parent(leftmost-leaf(N)), N). In the algorithm, we assume the existence of the following simple functions whose semantics are straightforward: left-child(), right-child(), response-time(), parent(), and leftmost-leaf().

```
Hybrid(N1, N2) {
  loop
    L1 = left-child(N1);
    R1 = right-child(N1);
    L2 = left-child(N2);
    R2 = right-child(N2);
    if (response-time(L1) <= response-time(R2)) {
      LAN = Find-LAN(N1);
      if (LAN != null) {
        Transform(N1, LAN);
        if (response-time(NJN) >
            response-time(sibling((NJN)))
          Hybrid(parent(leftmost-leaf(NJN)), NJN);
        N1 = parent(N1); /* bottom-up */
      }
    }
    else {
      LAN = Find-LAN(N2);
      if (LAN != null) {
        Transform(N2, LAN);
        if (response-time(NJN) >
            response-time(sibling((NJN)))
          Hybrid(parent(leftmost-leaf(NJN)), NJN);
      }
      N2 = L2; /* top-down */
    }
  until (N1 = N2);
}
```

Figure 9: Hybrid Algorithm

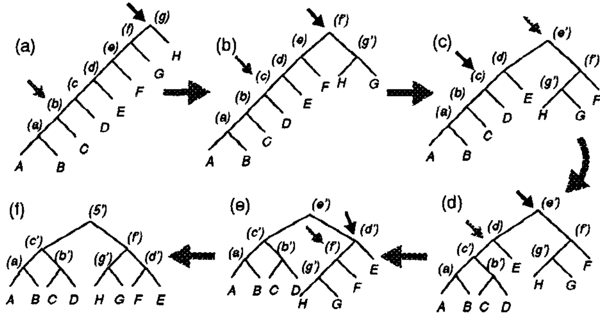

Figure 10: Example of hybrid transformation

In the hybrid approach, two sequences of transformations happen concurrently: one proceeds top down from the root, and the other proceeds bottom up from the leaf. Correspondingly, there are two UANs in the algorithm: N_1 for the bottom-up transformation and N_2 for the top-down transformation. Initially, N_1 is set to the grandparent of the leftmost leaf node of the original join tree, and N_2 is set to the root. At any given time, at most one UAN is active. The active UAN is chosen between the two UANs by comparing their child subtrees: N_1 is chosen if the response time of its left child subtree is smaller than that of N_2's right child subtree, and N_2 is chosen otherwise. If the response times are about the same, the UAN whose child subtree is unbalanced will be chosen.

A basic transformation is then identified and applied to the active UAN. The transformation will make the child subtree of the active UAN node bigger. The purpose is to keep similar response times for the two subtrees. The algorithm stops when the two UANs meet. When this happens, the UANs become the new root for the balanced join tree, and the two subtrees are child subtrees of the new root.

The algorithm also tries to keep both the left child subtree of N_1 and the right child subtree of N_2 balanced after each transformation. According to the bottom-up protocol, the NJN resulting from the transformations to N_1 is already balanced. We need to balance the NJN after each basic transformation to N_2 (i.e., the top-down UAN). Again, the NJN may be unbalanced as LAN's right child subtree could be an arbitrarily large subtree resulted from previous transformations (e.g., Figure 5). In the original top-down algorithm, NJNs are balanced after the load has been evenly distributed for the new root.

The basic top-down approach can also been extended in the hybrid algorithm to allow basic transformations that do not improve overall query response time but facilitate future transformations. Similar to the bottom-up approach, transformations are allowed only if UAN's right child subtree is unbalanced. The hope is that subsequent transformations that balance the NJN will improve the overall query response time.

4.3.2 Example

Figure 10 shows how the left deep join tree in Figure 5.a and Figure 7.a is transformed into a balanced join tree using the hybrid approach. Again, it only shows steps in which basic transformations have taken place. Two UANs are highlighted using arrows and the active UAN is highlighted using a solid arrow. Join nodes (c) and (g) are chosen to be the initial two UANs. Join node (g) is the first active UAN as leaf node H is smaller than the join node (b). Join node (f') becomes the new UAN after the transformation and remains active.

In Figure 10.d, join node (e') is the active UAN, not because its child subtree rooted by (f') has a smaller response time than the subtree rooted by (c'), but because the subtree rooted by (f') is unbalanced. Please note that the transformation does not reduce the overall query response time. As shown in Figure 10.e and Figure 10.f, the subsequent transformation does reduce the overall response time.

4.3.3 Discussion

The hybrid algorithm has similar complexity as the top-down and bottom-up approaches. For example, in the worst case (described in Subsection 4.1) where the leftmost leaf node dominates the total cost at each step, the hybrid algorithm invokes the same number of basic transformations, since the bottom-up UAN will never be activated.

The major advantages of the hybrid approach over the top-down and bottom-up approaches is that it generally produces more balanced bushy join trees. Since redistribution of loads is based on the response time of the balanced child subtrees, the problem described in Figure 6 will not occur. The hybrid approach is also superior to the bottom-up approach as loads are evenly distributed for the new root. Figure 11.a and Figure 11.b show the results of hybrid transformation on the left deep join trees in Figure 6.a and Figure 8.a, respectively. They are more balanced than those generated by the top-down and bottom-up approaches (Figure 6.c and Figure 8.b).

Again, the hybrid approach does not guarantee the optimality of transformed join trees. It is also possible that the top-down or bottom-up approach produce more balanced bushy trees for some left deep join trees. This is possible, for example, for left deep join trees that can be perfectly balanced by the bottom-up approach, but not by the top-down approach. The

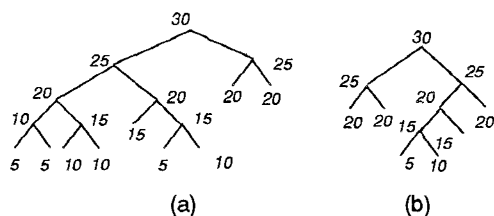

Figure 11: Hybrid vs. top-down and bottom-up

hybrid approach may not produce the best result due to the partially unbalanced right half of the transformed bushy tree that is generated by the top-down approach. The hybrid approach is generally considered to be better than both the top-down and bottom-up approaches because it avoids extreme cases that may be produced by the other two, and because it generally produces better results for most join trees.

5 Related Issues

5.1 Working with Existing Optimizers

So far, we have described transformations that are useful in reducing the response times of multidatabase queries. We now discuss how the technique can be integrated with existing query optimizers.

There are two possible ways that an existing query optimizer can be extended. In the two-phase approach, multidatabase queries are first optimized with respect to total cost using the traditional optimization strategy; then the resulting left deep join trees are improved (with respect to response time) using the algorithms described above. The advantage of the two-phase approach is that no modification to the existing optimizers is needed.

The second approach is an integrated one which concurrently performs optimization with respect to total cost and tree transformation to improve response time. More specifically, for each subset of join tables, we first obtain the optimal (left deep join) subplan with respect to total cost using the traditional approach. This subplan is then balanced using the proposed transformations. The improved subplan and the associated cost (both total cost and response time) are then stored into the subplan table for future reference. The advantage of this approach is that optimization and transformation may benefit each other; therefore it may result in better execution plans. The disadvantage is that it requires modifying an existing optimization and it may increase optimization time.

5.2 Relaxing Basic Transformations

The basic principle behind the proposed strategy is to keep the basic transformation simple.

An alternative is to allow complex basic transformations. There is clearly a trade-off between the two approaches. In general, the more flexible a basic transformation, the fewer steps it takes to balance a query tree; on the other hand, the more overhead it incurs in a single transformation. There are two reasons for this: the transformation affects a larger segment, and the selection of LANs becomes more complicated.

Another simple extension of the basic transformation is to allow limited increase in total cost. Such a transformation can be justified if (1) it yields a big improvement in response time, or (2) it facilitates subsequent transformations. The specification of the allowance can be either absolute or relative (with respect to improvement in response time). Clearly, the relaxation should be done in a controlled way, especially for the second purpose.

5.3 Pruning Non-Cost-Improving Basic Transformations

Heuristics can be used to prevent transformations that are unlikely to be cost improving, thereby reducing transformation overhead. For example, a basic transformation is very unlikely to improve the overall cost if there is no equality join predicate for the NJN (e.g., between nodes i and j in Figure 3). This is consistent with the traditional optimization that delays cross products as late as possible. We can actually go a little further by requiring that the size of the NJN be no greater than that of LAN's right child node. The intuition behind the heuristic is that a small join result may benefit the subsequent joins. In Figure 3, for example, if the join result of nodes i and j is smaller than node i, the join between the NJN and node $i+1$ could be cheaper to evaluate and produces a smaller result, which in turn benefits the subsequent join with node $i+2$, and so on.

6 Conclusion

In this paper, we have addressed an important issue in multidatabase query optimization, namely reducing response time for a total cost optimal multidatabase query execution plan. While previous work has considered optimization of query response time, the present paper is the first to outline techniques that are specially suited to multidatabase queries.

Our work is motivated by the following observations of multidatabase query execution:

- Nested loop join is less frequently used to implement global joins in a multidatabase query due to overhead incurred in each inner table lookup.
- Sort merge and hash joins cause large hierarchical delays in multidatabase systems due to the sequential nature of their implementation.

These features introduce additional difficulties for multidatabase query optimization. On the one hand, left deep join trees as generated by the traditional query optimizers for multidatabase queries are often suboptimal, especially with respect to query response time. On the other hand, it is very expensive for a multidatabase query optimizer to exhaustively search a large execution space that includes all bushy join trees. We address the problem by first optimizing multidatabase queries using the traditional optimization strategy, and then improving response time of the left deep join trees obtained using simple tree transformations. Compared to existing strategies, the proposed approach has the advantages of guaranteed minimum total cost, improved response time, and low optimization overhead.

References

[BTY84] D. Brill, M. Templeton and C. Yu. Distributed Query Processing Strategies in Mermaid, A Frontend to Data Management Systems, In Proc. *IEEE Data Engineering*, Los Angelos, CA, 1984.

[Chen90] A. Chen. Outerjoin Optimization in Multidatabase Systems, In Proc. *Distributed and Parallel Database Systems*, Dublin, Ireland, 1990.

[Daya83] U. Dayal. Processing Queries Over Generalization Hierarchies in a Multidatabase System, In Proc. *VLDB*, 1983.

[Daya85] U. Dayal. Query Processing in Multidatabase System, In *Query Processing in Database Systems* (Kim, Batory and Reiner (eds.), 1985), Springer Verlag.

[DH84] U. Dayal and H. Hwang. View Definition and Generalization for Database Integration in Multibase: A System for Heterogeneous Distributed Databases, In *IEEE Tran. on Software Engineering*, Vol. 10, No. 6, 1984.

[DKS92] W. Du, R. Krishinamurthy and M. Shan. Query Optimization in a Heterogeneous DBMS. In Proc. *VLDB*, Vancouver, Canada, 1992.

[DSD94A] W. Du, M. Shan and J. Davis. Optimization and Execution Strategy for Multidatabase Queries, Technical Report *HPL-94-74*, Hewlett-Packard Labs., 1994.

[DSD94B] W. Du, M. Shan and U. Dayal. Reducing Multidatabase Query Response Time By Tree Balancing, *DTD Technical Report*, Hewlett-Packard Labs., 1994.

[Hong92] W. Hong. Exploiting Inter-Operation Parallelism in XPRS, In Proc. *ACM SIGMOD*, San Diego, CA, 1992.

[HS93] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS, In *Distributed and Parallel Databases*, Vol. 1, No. 1, 1993.

[IK90] Y. Ioannidis and Y. Kang. Randomized Algorithms for Optimizing Large Join Queries, In Proc. *ACM SIGMOD*, Atlantic City, NJ, 1990.

[LMH+85] G. Lohman, C. Mohan, L. Haas, B. Lindsey and P. Selinger. Query Processing in R*, In *Query Processing in Database Systems* (Kim, Batory and Reiner (eds.), 1985), Springer Verlag.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price. Access Path Selection in a Database Management System, In Proc. *ACM SIGMOD*, 1979.

[SAD+94] M. Shan, R. Ahmed, J. Davis, W. Du, and W. Kent. The Pegasus Project - A Heterogeneous Information Management System, In *Modern Information Computer* (Kim ed.), Addison-Wesley, 1994.

[SL90] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, In Proc. *ACM Computing Surveys*, Vol. 22, No. 3, 1990.