Memory Allocation Strategies for Complex Decision Support Queries *

Biswadeep Nag
bng@cs.wisc.edu
Computer Sciences Department
University of Wisconsin-Madison

David J. DeWitt
dewitt@cs.wisc.edu
Computer Sciences Department
University of Wisconsin-Madison

Abstract

Complex decision support queries, as exemplified by the TPC-D benchmark, make heavy use of join, aggregation and sorting operations. These operations can be very memory intensive and it is unlikely (particularly in a multi-user environment) that a database system will always have enough memory to completely satisfy the demands of all concurrently running operators. In this paper we propose, analyze and evaluate four different algorithms that distribute available memory among operators in ways that satisfy all operator requirements and concurrent schedulability constraints. One of these algorithms, based on linear programming, is also optimal in that it is guaranteed to produce the best query response time. We also show how these algorithms can be extended to work on shared-nothing parallel database systems. All the proposed strategies are completely general and can be used with any extended relational operations that show a linear improvement in performance with increased memory availability.

1 Introduction

The growing trend towards building large data warehouses that can be used to answer complex decision support queries [1] has been driven by a wealth of research in designing efficient algorithms for common relational operations like join, sort, and aggregation. To ensure acceptable performance on multi-gigabyte databases these algorithms usually require a large amount of main memory which, if intermediate I/O is to be avoided altogether, is proportional to the size of the input for a join and proportional to the size of the output for an aggregate/group-by operation. It is, however, unrealistic to expect that it will always be possible to completely satisfy the total memory requirements of a query that contains several joins and aggregates. This is especially true in a multi-user environment where many such complex decision support queries are expected to run concurrently. Given the situation that each such query can only be allotted a fraction of the total main memory and that such an allocation is unlikely to satisfy the demands of all operators, we are faced with the problem of deciding how to divide the memory that is available among the different operators in a query. In this paper, we show how to choose an allocation that is guaranteed to produce the best overall execution time for the query. Our scheme is very general and incorporates all possible query plan tree shapes and all common relational operations.

*This work was supported by DARPA through grant number F30602-97-2-0247.

An idea of the complexity of queries that are important in decision support applications can be had from the TPC-D benchmark [16]. Out of the 17 (non-update) TPC-D queries, 6 queries contain at least 3 joins, 13 contain a group-by aggregate and 13 contain an order by operation. To ensure adequate response times for these queries, most systems make use of intra-query parallelism which can take one or more of the following forms: (1) Partitioned Parallelism (2) Pipelined Parallelism and (3) Independent Parallelism. Although the benefits of employing parallelism are widely known, it has also been noticed in [13] that increasing the degree of parallelism also increases the demand for memory. For example, a longer operator pipeline can be constructed in right-deep trees than in left-deep trees, but this causes greater resource contention because more join operators run concurrently. A glance at the currently available TPC-D results [17] reveals that only a handful of reported results uses more than 1 query stream. This indicates that the problem of allocating resources for complex queries under heavy load conditions is far from being solved.

In this paper we examine the question of how to allocate memory to the different concurrently running operators of a query in a way that will minimize the total query response time. Related work in this area is surveyed in Section 2. In Section 3 we formulate the memory allocation problem and enumerate the constraints that must be obeyed by any viable solution. In Section 4 we present a set of algorithms that provide solutions to this problem. One of these algorithms, LINEAR, is guaranteed to produce an optimal solution. We also present SMALL, a heuristic algorithm whose performance is close to the LINEAR algorithm but which is much simpler to implement. Section 5 presents experimental results, followed by a discussion in Section 6 of how the algorithms can be extended to shared-nothing parallel database systems. Finally Section 7 contains our conclusions.

2 Related Work

Previous research on the question of resource management for complex queries can be classified into three categories. In the first category are studies dealing with the interaction of query optimization and memory availability in a parallel environment. [15] provides a heuristic search strategy for constructing a query plan tree that also considers resource availability. [2] proposes the use of segmented right-deep trees that combine the deep pipelining offered by right-deep trees with the flexibility of bushy trees. The focus of all these algorithms is on generating the best query plan tree using the most efficient search strategy. The addition of resource considerations into query optimization usually results in an explosion of the plan space. To prune the search space, each of these algorithms assumes that there is always enough memory to completely satisfy the requirement of each join. In reality, however, this assumption is much too restrictive since most relational operators are *malleable* in the sense that they can operate in a range of memory allocations between their minimum and maximum requirements. The task of determining the best query plan which takes into account this malleability of operators is generally considered a hard optimization problem [4]. In fact, this problem is rather similar to that of query optimization for parallel database systems. [5] mentions that one can either employ a one phase (optimization combined with parallelization) or a two phase (optimization followed by parallelization) strategy. The second method is definitely much easier to implement and has not been conclusively proved to be inferior to the first. In this paper, we also adopt a two phase approach. The optimizer first produces a query plan that assumes sufficient memory availability and the memory manager then generates the memory allocation for each operator depending on the exact amount of memory available. There are a number of reasons why this two phase approach might actually work better. In addition to reducing the complexity of query optimization, this approach cleanly separates run-time aspects of query execution from decisions taken at compile-time. Not only is it difficult to incorporate run-time state information such as system load and memory availability into the query optimizer, but it also leads to instability in the plans generated. It is also much easier to adapt the two phase approach to shared-nothing parallel database systems.

The second area of related research concentrates on the optimal scheduling of operators and queries in a parallel system. [7] and [12] both propose strategies for determining the degree of parallelism and choosing the nodes on which the operators of the query should run. [4] also provides scheduling algorithms based on multi-dimensional bin packing that take into account both time-shared (CPU, disk, network) and space-shared (memory) resources. Despite the immense amount of work done in this field, there have been very few studies dealing with the execution of joins under conditions of insufficient memory availability. Most of the above algorithms restrict their attention to only those nodes that have enough memory to satisfy the maximum memory requirements of all joins that might be scheduled at that node. In the event that not enough memory is available to completely satisfy at least two joins per query, most algorithms resort to sequential execution.

The third category of papers actually considers a multi-query environment where all queries may not get their maximum memory requirement. [8] examines this problem in the context of real-time database systems where queries may need to be started in the absence of sufficient memory so that they can meet their deadlines. [3] and [19] describe how memory might be pre-empted from hash join operations during periods of high memory demand. [6] divides queries into three categories depending on their memory requirements and provides allocation algorithms for these three classes of queries based on a fairness metric. [18] offers a useful characterization of joins based on their *return on consumption* and shows that only joins with a return on consumption above a certain threshold will benefit from being given their maximum memory requirement. One of the heuristic algorithms presented in this paper (SMALL) is based on this return on consumption metric. A shortcoming of all these studies is that they deal only with single join queries and do not consider queries with multiple join and aggregate operations — something that is extremely common in decision support.

3 Definitions and Problem Formulation

Many commercial database systems base their optimizers on the System R optimizer [14] which generates only left-deep trees for a query with several joins (such as the one in Fig.1). The most common scheduling strategy [13] for such a tree is to break the tree up into a number of concurrently schedulable units which we call *shelves* (Fig.2). Operators within a shelf are pipelined (light horizontal edges) while there are blocking dependencies [5] (dark vertical edges) between at least one pair of operators in adjacent shelves. In this paper we concentrate exclusively on the hybrid hash implementations of join and aggregate operations. In the case of hybrid hash join, there is a blocking dependency between the *build* (B_i) and *probe* (P_i) phases of the join and so these two phases must reside in different shelves. From now on, we will use the terms operators and operator phases interchangeably. The meaning should be clear from the context.

We assume a multi-user environment where a certain active query is allotted a maximum of M pages of main memory. M is usually determined by an algorithm [6][18] that considers the memory requirement of the query, the total amount of memory in the system and the priority of the query as decided by the database administrator. This is a typical way in which commercial database systems currently manage resources [11]. There may be other load control strategies which influence the number of active queries and also the value of M for a particular query. These factors neither influence the design of our algorithms nor affect our analysis, hence we do not consider them further. The following definitions formalize the notions of query, shelf and schedule.

Definition 3.1 *A query Q is a set of operators (actually operator phases for two phase operators like hash join), $\{OP_1, OP_2, \ldots, OP_{N(OP,Q)}\}$ where $N(OP,Q)$ is the number of operators in the query.*

Definition 3.2 *A shelf S is a set of operators (or phases) $\{OP_{S,1}, OP_{S,2}, \ldots, OP_{S,N(OP,S)}\}$ all of which must be scheduled concurrently, in one or more pipelines. The number of operators in the shelf S is given by $N(OP,S)$. $OP_{S_j,i}$ denotes the ith operator in shelf S_j.*

Definition 3.3 *A schedule Ψ for a query Q is a partitioning of Q into shelves, i.e. $\Psi = \{S_1, S_2, \ldots, S_{N(S,Q)}\}$ such that $\bigcup_{i=1}^{N(S,Q)} S_i = Q$ and $S_i \cap S_j = \phi$ if $i \neq j$. The shelves are numbered in order of execution and there are $N(S,Q)$ shelves in the schedule for the query.*

For example, Fig. 2 shows a possible schedule for the query in Fig. 1. We now formalize the nature of the output of a memory allocation algorithm.

Definition 3.4 *The memory allocation $\vec{M}$ for a query Q is*

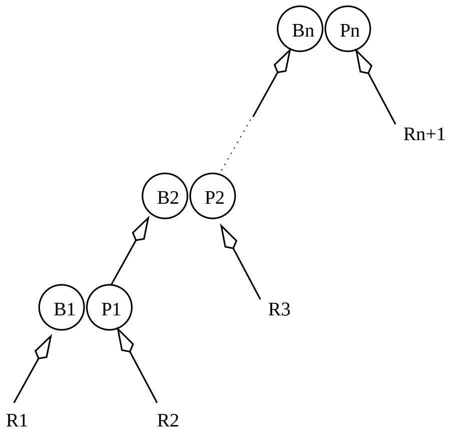

Figure 1: A Left-deep Query Tree

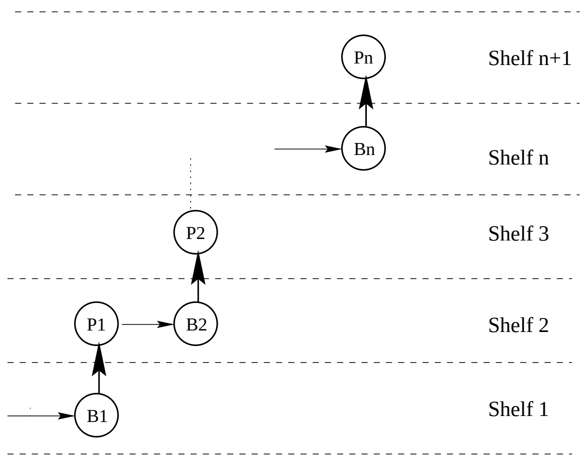

Figure 2: Query broken up into Shelves

a vector of length $N(OP, Q)$ where $M[k]$ denotes the number of pages of memory allocated to OP_k.

Definition 3.5 *A* valid operator memory allocation $\vec{M}_{OP}$ *is a memory allocation that satisfies all operator constraints, i.e.* $\forall_{k=1}^{N(OP,Q)} M_{min}[k] \leq M_{OP}[k] \leq M_{max}[k]$, *where $M_{min}[k]$ and $M_{max}[k]$ denote the minimum and maximum memory requirements of OP_k respectively.*

Note that these *operator constraints* must be satisfied regardless of the shape of the query plan tree and the exact schedule Ψ produced by the scheduling algorithm. Since a shelf is a concurrently schedulable set of operators, we obtain the following set of *shelf constraints*:

$$\forall_{j=1}^{N(S,Q)}((\sum_{k=1}^{N(OP,S_j)} M[OP_{S_j,k}]) \leq M) \tag{1}$$

In other words, for any shelf, the sum total of the memory allocations for the operators executing in the shelf must not exceed the total allowable memory. There are two implicit assumptions being made here. One is that the memory allowed for each shelf is the same as the total allowable memory for the query. This is true because no two shelves execute concurrently. The second assumption is that the memory consumed by the build phase of a join is equal to the memory consumed by the probe phase of the join and neither of these change during the execution of the query. This is because the hash table built by the build phase of the hash join is exactly the same hash table that is probed during the probe phase of the join. Specifically, we are not preempting memory from the operators as suggested in [3][19].

Definition 3.6 *A* valid query memory allocation $\vec{M}_{val}$ *is a valid operator memory allocation that also satisfies all shelf constraints.*

Definition 3.7 *For a given scheduler Ψ for a query Q, an* optimal query memory allocation $\vec{M}_{opt}$ *is a valid query memory allocation that produces the lowest query response time.*

In the next section we explore four strategies for finding valid query memory allocations, one of which is also guaranteed to produce an optimal query memory allocation given the constraints mentioned above.

Algorithm EQUAL;
$\forall_{i=1}^{N(OP,Q)} M[i] = 0$;
  /* No memory allotted initially */

Foreach shelf S_j $(1 \leq j \leq N(S,Q))$
  $M_{excess} = M - \sum_{i=1}^{N(OP,S_j)} M_{min}[OP_{S_j,i}]$;
    /* Excess memory that can be distributed */

  Foreach operator $OP_{S_j,i}$ $(1 \leq i \leq N(OP,S_j))$
    $M_{allot} = M_{excess}/(N(OP,S_j) - i + 1)$;
      /* Divide excess memory equally */
    $M_{temp} = \min(M_{min}[i] + M_{allot}, M_{max}[i])$;
      /* Must satisfy operator constraints */

    If $(M[i])$ /* Memory allocated in previous shelf */
      $M[i] = \min(M[i], M_{temp})$;
      /* Can't violate constraints of previous shelf */
    Else
      $M[i] = M_{temp}$;
    $M_{excess} -= M[i] - M_{min}[i]$
      /* If OP_i didn't use its share, give it to others */
Return $\vec{M}$;

Figure 3: The Equal Algorithm

4 Memory Allocation Strategies

4.1 The EQUAL Algorithm

The first naive strategy that comes to mind is to divide the available memory equally among all the operators in a shelf while insuring that all operator constraints are satisfied. Additionally we must ensure that while deciding the memory allocation of one shelf we do not violate the constraints of other shelves. This is done by the algorithm given in Fig. 3. Though the EQUAL algorithm produces a valid query memory allocation, there is no assurance of it producing a memory allocation anywhere near optimal. The only reason it is mentioned in detail here is because two other algorithms we will discuss have the same general structure except that they make a more intelligent choice of which operator to favor in each shelf.

$$\text{I/O Cost} = T_{I/O}(3-2f)(|R|+|S|)$$

$$\begin{aligned}\text{CPU Cost} = (1/\text{MIPS}) * [&I_{I/O}(3-2f)(|R|+|S|) \\ &+I_{move}(3-2f)(\{R\}+\{S\}) + I_{join}\psi\{R\}\{S\} \\ &+I_{hash}(2-f)(\{R\}+\{S\}) + I_{search}(\{R\}+\{S\})]\end{aligned}$$

Figure 4: Hybrid Hash Join Cost Model [18]

4.2 The BENEFIT Algorithm

Like the EQUAL algorithm, the BENEFIT algorithm also generates a valid query memory allocation but favors operators that will actually *benefit* the most from additional memory. This is done by calculating a *benefit function* based on an analysis of operator costs. For now we will illustrate the algorithm using the hash join and hash aggregation operations but this idea can be extended to any operator for which such a constant benefit function can be calculated.

Definition 4.1 *The* benefit function $B(OP)$ *for an operator* OP *is the incremental reduction in cost achieved per additional page of memory allocated.*

Let R and S be the relations being joined by the hybrid hash join algorithm and let $|R|$ and $|S|$ denote the respective number of pages of the two relations while the number of tuples is denoted by $\{R\}$ and $\{S\}$ respectively. Fig. 4 describes the costs associated with the hybrid hash join operation in terms of these parameters (assuming R is the build relation and S is the probe relation). $f = M/|R|$ represents the fraction of the build and probe relations that can be kept in memory, the various I's represent the instruction path lengths for different operations, $T_{I/O}$ is the average I/O service time, *MIPS* is the CPU speed and ψ is the join selectivity. An examination of this cost model reveals that only some of the terms (those which contain f) depend on the actual memory allocation. These terms serve to reduce the join cost as more memory is made available. Hence, by grouping together the constant terms under C_{const} and the memory dependent terms under C_m, the total cost (including both CPU and I/O) of a hybrid hash join can be written as: $C_t = C_{const} - C_m$ At this point we note that the expression for C_m includes quite a few system dependent parameters (such as I_{hash}, I_{move}, *MIPS* etc.) that are constant across all queries. If we regard only the query dependent parameters as independent variables, then we can express C_m in terms of them as:

$$\begin{aligned} C_m &= \frac{M}{|R|} * (C_J + C_{J,|R|} * |R| + C_{J,\{R\}} * \{R\} \\ &\quad + C_{J,|S|} * |S| + C_{J,\{S\}} * \{S\}) \end{aligned} \tag{2}$$

The constants $C_{J,|R|}, C_{J,\{R\}}, C_{J,|S|}, C_{J,\{S\}}$ represent the lumping together of costs that are proportional to $|R|$, $\{R\}$, $|S|$, $\{S\}$ respectively. The BENEFIT algorithm recognizes that the key to deciding which join will benefit more from memory is determined by the C_m term which is proportional to the memory allocation. The following result is obtained directly from equation 2.

Result 4.1 *The benefit function of a join* $J = R \bowtie S$ *is given by*

$$\begin{aligned} B(J) &= \frac{1}{|R|} * (C_J + C_{J,|R|} * |R| + C_{J,\{R\}} * \{R\} \\ &\quad + C_{J,|S|} * |S| + C_{J,\{S\}} * \{S\}) = C_m/M \end{aligned} \tag{3}$$

A similar investigation of the hash aggregation (group-by) operation yields the following result.

Result 4.2 *The benefit function of an aggregate operation* $AG(R)$ *is given by*

$$B(AG) = \frac{1}{(|G|+|A|)} * (C_A + C_{A,|R|} * |R| + C_{A,\{R\}} * \{R\}) \tag{4}$$

where $|G|$ *and* $|A|$ *are the total size of all group-by and aggregate attributes respectively and* $C_A, C_{A,|R|}$ *and* $C_{A,\{R\}}$ *are system dependent constants.*

The BENEFIT algorithm gives more memory to joins with a larger benefit function since these are the operators whose cost will reduce the most because of additional memory. The design of the algorithm is similar to that of the EQUAL algorithm except that it does not distribute excess memory equally. Instead, it sorts the operators in a shelf based on the values of their benefit function and then traverses this sorted list in descending order giving to an operator at each stage as much memory as possible without violating its operator and shelf constraints.

4.3 The LINEAR Algorithm

The drawback of the EQUAL and BENEFIT algorithms is that these algorithms look at only one shelf at a time and do not consider the global situation. Hence they are unlikely to reach the global optimum. This is illustrated by the following example.

Consider a query Q with four joins J_1, J_2, J_3 and J_4 that are divided into five shelves S_1, S_2, S_3, S_4 and S_5 such that join J_i straddles the shelves S_i and S_{i+1}. Notice that the decision taken during the processing of shelf S_2 (i.e. the choice of whether to favor join J_1 or J_2) will influence the choice in shelf S_4. If join J_1 is given less memory, then J_2 will probably get more memory (unless there is enough memory to accommodate both the joins) and consequently J_3 might get less memory and J_4 more memory and so on. The problem occurs when J_3 has a higher benefit function than J_4, or when J_4 has a very small memory requirement. In that case, the algorithms that we have discussed thus far might make wrong decisions in shelf S_4 and produce memory allocations that are sub-optimal or even leave memory unused in some shelves. The basic drawback of these algorithms is that they have only a per shelf view of the query.

The LINEAR algorithm tries to remedy just that. Instead of allocating memory on a shelf-by-shelf basis, it makes its decision in one global optimization step. The key observation behind the design of the LINEAR algorithm is that the hash join and hash aggregation operations both show a linear improvement in execution time with increased memory allocation. This is obvious from cost formula (2) where the memory dependent terms are all linear in M. We have also experimentally verified this fact by running many hash join and aggregation operations in the Paradise database system. Fig. 5 shows how the execution time of a single hash join operator varies as the memory allocation is increased from its minimum requirement to its maximum demand (which in this case is 20 MB). Fig. 6 illustrates the results of a similar experiment with an aggregate/group-by operation that produces 20,000 groups. The reader will notice that the *benefit function* of an operator is nothing but the negative of the

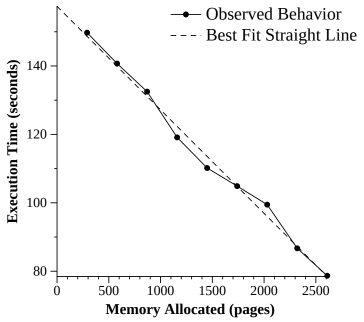

Figure 5: Memory Dependence of Hash Join

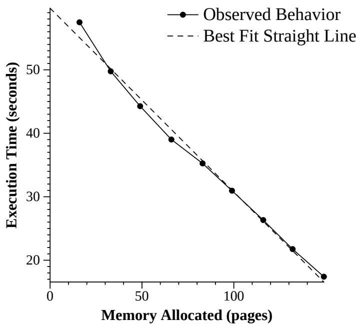

Figure 6: Memory Dependence of Group By

slope of these curves. We can now extend the notion of a benefit function to the whole query.

Definition 4.2 *The benefit function of a query $Q = \{OP_1, OP_2, \ldots OP_{N(OP,Q)}\}$ with memory allocation $\vec{M}$ is given by*

$$B(Q, \vec{M}) = \sum_{i=1}^{N(OP,Q)} B(OP_i) * M[OP_i] \quad (5)$$

We can now view the problem of query memory allocation as a *linear programming* problem where the *query benefit function* (which is linear in the $M[i]$s) serves as the *objective function*, since a memory allocation that maximizes the value of the query benefit function will minimize the total cost of the query. The set of constraints (which are also linear in $M[i]$s) is the union of the set of $2N(OP, Q)$ operator constraints of the form $M[i] \geq M_{min}[i]$ and $M[i] \leq M_{max}[i]$ and the set of $N(S, Q)$ shelf constraints which take the form $\forall_{j=1}^{N(S,Q)} \sum_{i=1}^{N(OP,S_j)} M[OP_{S_j,i}] \leq M$.

The LINEAR algorithm calculates $M_{min}[i]$, $M_{max}[i]$ and $B(OP_i)$ for each operator and then feeds this information into a linear programming problem solver to try and maximize $B(Q, \vec{M})$. In particular, it uses the *simplex method* which will find a solution (if one exists) that satisfies all given constraints and maximizes the value of the objective function [10]. Thus the LINEAR algorithm is guaranteed to find an optimal query memory allocation that minimizes the query response time.

4.4 The SMALL Algorithm

Though the LINEAR algorithm is guaranteed to produce an optimal query memory allocation, it is slightly cumbersome to implement because it requires as input a number of system dependent parameters ($C_{J,|R|}, C_{J,\{R\}}$ etc.) which can only be obtained by calibrating the system (section 5.1). The accuracy of the memory allocation produced is also sensitive to the accuracy of the estimates of the sizes of the intermediate result relations. We therefore found it necessary to design a heuristic algorithm which performs almost as well as LINEAR but is considerably simpler to implement and does not require very accurate estimates.

The SMALL algorithm generates a valid query memory allocation in a way similar to the EQUAL algorithm, but it deals with the *excess* memory (the memory left over after giving all the operators in a shelf their minimum requirement) in a different way. It preferentially allocates more memory to the operators that have a smaller demand. This is done by sorting the operators in a shelf in the order of their $M_{max}[i]$ and offering M_{excess} to these operators in ascending order. The rationale behind this heuristic is based on the results in [18] where it is shown that smaller joins generally have a better return on consumption and are likely to benefit more from additional allocation of memory. Support for this heuristic is also obtained from equation (3) where it can be seen that joins with small $M_{max}[i]$ (i.e. small $|R|$) will generally have a larger benefit function.

The SMALL algorithm enjoys the advantage of being much simpler to implement compared to the BENEFIT and LINEAR algorithms since it only requires knowledge of $M_{min}[i]$ and $M_{max}[i]$ for each operator. However, as we shall see in section 5, it produces memory allocations whose performance is surpassed only by the optimal memory allocations that are generated by the LINEAR algorithm.

5 Experimental Results

We now describe the results of some experiments conducted to evaluate the performance of the EQUAL, BENEFIT, LINEAR and SMALL algorithms. The experimental vehicle for our study was the Paradise parallel, object-relational database system [9]. In addition to supporting standard SQL-92 data types and most common relational operations, Paradise has been designed to handle user-defined ADTs and domain specific operations on them. In this study however, we are only concerned with relational operations like joins and aggregates on standard data types such as integers, reals and strings.

5.1 The Calibration Phase

Implementing the EQUAL and SMALL algorithms was simple since we only needed to know the minimum and maximum memory requirements of the operators. For the BENEFIT and LINEAR algorithms however, we needed a way to calculate the operator and query benefit functions. The query dependent parameters, such as $|R|$, $\{R\}$, $|S|$, $\{S\}$, $|G|$, $|A|$, were readily available from the query optimizer but we still needed to find proper values for the system-dependent constants C_J, $C_{J,|R|}$, $C_{J,\{R\}}$, $C_{J,|S|}$, $C_{J,\{S\}}$, C_A, etc. Most of these constants are also a combination of seemingly disparate CPU and I/O costs and it was necessary to find a reliable way of accurately estimating their values.

Rather than making inexact estimates of path lengths or obtaining approximate values from the literature, we decided to *measure* these system dependent parameters by *calibrating* Paradise. We ran a large number of different hash join and aggregate queries and for each query noted the variation in execution time with different memory allocations. We then used the *least squares* method to find the best fit straight line for each query. The negative of the slope of this line equals the value of the benefit function for the operator. In theory, we only needed to run 5 queries to solve for the 5 constants in the hash join benefit function. In practice however, due to the presence of inevitable, random experimental errors, using only 5 queries leads to instability in the values obtained. This problem was solved by running more experiments (we ran 15 queries) and again using the least squares approach to find the best values for these constants. We did similar calibration runs to determine the hash aggregate benefit function which had 3 unknown constants.

5.2 The Experimentation Phase

The experimental system was a 200 MHz Pentium Pro machine running Solaris 2.5.1 with 128 MB main memory and a 1 GB Quantum Fireball disk. Since this was a uniprocessor system, Paradise utilized only pipelined parallelism. Extension of the memory management algorithms to handle partitioned parallelism is discussed in section 6. We ran three sets of experiments with 4, 6 and 8 tables respectively. Each table was randomly generated with integer and string attributes making up 232 byte records. The table size was uniformly distributed with a mean of 300,000 tuples. The size of the database buffer pool was 32 MB. We intentionally chose not to have a larger buffer pool so that we could model a system with memory constraints and this allowed us to observe the differences in performance among the various memory allocation strategies.

Each i table experiment had $i-1$ joins and one aggregate/group-by operator. This workload is typical of decision support queries [16]. The number of tuples in the result of a join was always equal to the number of tuples in the smaller input relation. The size of each result tuple was equal to the sum of the sizes of the tuples in the two input relations. The use of these assumptions enabled us to create wide variations in the distribution and sizes of the input relations to the queries yet at the same time allowed us to accurately estimate the characteristics of the intermediate result relations. Each experiment set contained 10 randomly generated queries (random with regard to input and output relation sizes) and each query was repeated 5 times. The database buffers were flushed before each run. Since the execution times of the different queries were widely different, we used the *geometric mean* to summarize the results. Figs. 7 to 12 show the outcomes of these experiments and an analysis of the results follows. The reported times include the time taken to make the memory allocation decisions, which we found to be in the order of milliseconds, even for the LINEAR algorithm. This indicates that the running time of the memory management algorithms is insignificant when compared to the execution time of complex queries.

5.3 Analysis of Results

As expected, the LINEAR algorithm performs best by producing the lowest query execution times for all three sets of experiments (Figs. 7, 8, 9). The heuristic algorithm SMALL also does remarkably well, followed by the EQUAL algorithm. Surprisingly, the BENEFIT algorithm has the worst performance and we discuss the possible reasons for this below. Another interesting thing to note from Figs. 7, 8 and 9 is that the query benefit function shows a negative correlation with the query execution time for all experiments and strategies. This shows that the query benefit function (Equation 5) is a good choice for the objective function since the algorithms that maximize the query benefit function also minimize the query execution time. To facilitate further analysis of the results, we will make use of the following definitions.

Definition 5.1 *The* memory utilization *of a query/memory allocation* $(Q, \vec{M})$ *is given by*

$$M_{util}(Q, \vec{M}) = \frac{\sum_{i=1}^{N(OP,Q)} M[OP_i]}{\sum_{j=1}^{N(S,Q)} M} \tag{6}$$

Definition 5.2 *The* unit benefit *of a query/memory allocation* $(Q, \vec{M})$ *is given by*

$$U(Q, \vec{M}) = \frac{B(Q, \vec{M})}{\sum_{i=1}^{N(OP,Q)} M[OP_i]} \tag{7}$$

Note that the query benefit function can be expressed as a function of these two quantities only:

$$B(Q, \vec{M}) = M_{util}(Q, \vec{M}) * U(Q, \vec{M}) * \sum_{j=1}^{N(S,Q)} M \tag{8}$$

since $\sum_{j=1}^{N(S,Q)} M$ is independent of $\vec{M}$.

Intuitively, an algorithm which has the best *memory utilization* makes the greatest use of available memory. On the other hand, the algorithm with the greatest *unit benefit* extracts the most benefit from each page of memory it actually uses. In other words, each memory allocation algorithm can be characterized in terms of these two metrics both of which contribute to a larger query benefit function. Figs. 10, 11, 12 show that there is a tradeoff between memory utilization and unit benefit. Algorithms like BENEFIT and SMALL which have high unit benefit also have lower memory utilization. On the other hand, the EQUAL algorithm has low unit benefit and high memory utilization; so much so, that it produces a higher value for the query benefit function as compared to the BENEFIT algorithm. The reason that the SMALL algorithm turns in a reasonable performance is because its memory utilization is not as low as the BENEFIT algorithm. It appears that the LINEAR algorithm strikes the best balance by trading off just the right amount of unit benefit to ensure the highest memory utilization among all algorithms. However the memory utilization and the unit benefit improve for all algorithms as the queries become more complex. We believe this is because, in our experiments, the differences between the values of the benefit functions of the individual operators become smaller and so the performance gap between the various algorithms diminishes as the number of joins in the query increases.

It turns out that all the empirically observed characteristics of the memory management algorithms can be explained in terms of their design. The BENEFIT algorithm employs a greedy strategy that always chooses the operator with the greatest benefit function. The problem occurs when there

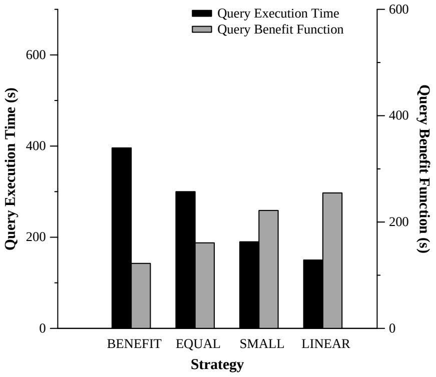

Figure 7: 4 table performance

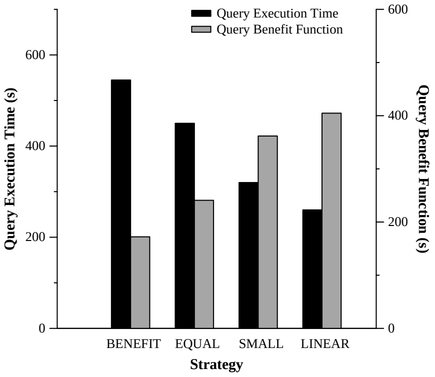

Figure 8: 6 table performance

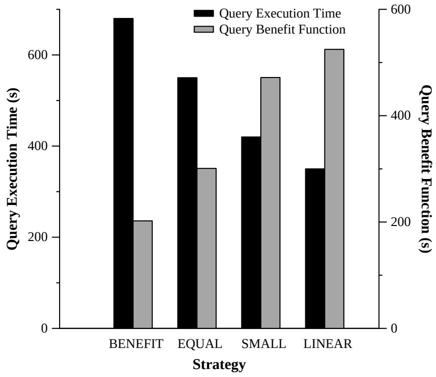

Figure 9: 8 table performance

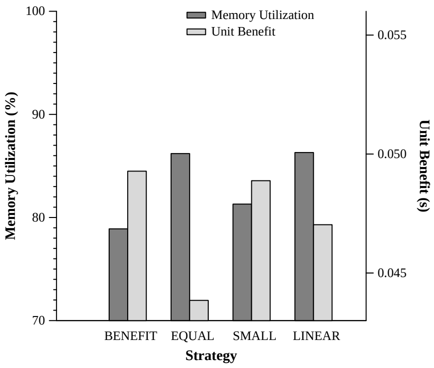

Figure 10: 4 table query analysis

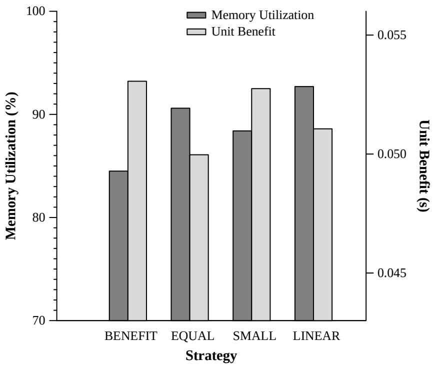

Figure 11: 6 table query analysis

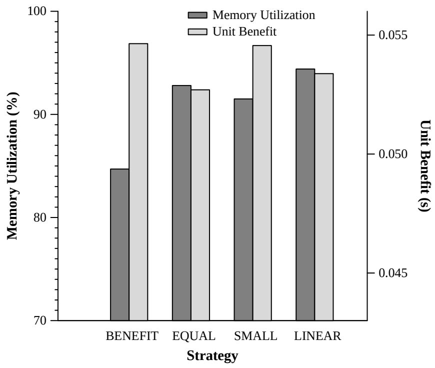

Figure 12: 8 table query analysis

is a shelf S_i where the "new" join operator J_{i+1} in that shelf has more benefit than the join J_i whose memory has already been decided while processing shelf S_{i-1}. It may so happen that the algorithm reduces $M[J_i]$ when J_i had already been granted more memory in shelf S_{i-1} (at the expense of join J_{i-1}). This might result in shelf S_{i-1} having low memory utilization. In the case of the SMALL algorithm, the optimizer already orders the joins roughly in the order of their memory requirements and so the algorithm usually favors join J_i rather than join J_{i+1} in shelf S_i since $M_{max}[J_i] < M_{max}[J_{i+1}]$. Therefore, memory utilization in shelf S_{i-1} does not drop. At the same time, joins with smaller values of $M_{max}[i]$ also tend to have greater benefit functions (because of the $|R|$ term in the denominator of equation (3)) and so the SMALL heuristic also generates a high unit benefit. This validates the choice of return on consumption of a join (proposed in [18]) as a feasible heuristic. The EQUAL algorithm does not look at unit benefit at all, but ensures that all the memory is always used (evenly), thus producing a high memory utilization. The LINEAR algorithm enjoys the best of both worlds. It is designed to choose operators with high benefit, yet at the same time not to allow memory utilization to fall to a low level. This results in the LINEAR algorithm producing the highest values for the query benefit function and the fastest query execution times. However, the SMALL algorithm turns out to be a close runner-up and may possibly be a better choice when a less complex implementation is desired.

6 Extension to Shared-Nothing Parallel Database Systems

While addressing the problem of memory managment in parallel databases, it should be noted that all the memory allocation algorithms presented in this paper will run on a single shared-memory system with little modification. This is because there is only one global shared pool of memory from which allocations are made and any operator executing on the SMP, regardless of which CPU it is using, can access that memory. As far as memory management is concerned, this situation is similar to a uniprocessor system. In the case of an MPP or a cluster of SMPs it is necessary to augment our algorithms to take into account the distributed nature of memory and we now show how the LINEAR algorithm can be generalized to handle this situation.

Let $N(nodes)$ be the number of nodes in the parallel system and let $M[i]$ represent the memory allotted to *each clone* of OP_i. $M_{min}[i]$ and $M_{max}[i]$ still denote the total minimum (and maximum) memory requirements of OP_i. In general, the number of clones of a particular operator (denoted by $N(OP_i, k)$) running on the various nodes might be different and similarly, the memory available for running the query $M_{sys}[k]$ might also be different on each node. We will assume that the number of clones to be activated for a particular operator as well as the choice of which nodes to run them on is decided by some scheduling algorithm such as the ones described in [4][7][12]. The task of the LINEAR algorithm is to find optimal values for $M[i]$. If $M[i]$ pages of memory are allotted to each operator clone, we obtain the following

$2N(OP,Q)$ operator constraints:

$$\forall_{i=1}^{N(OP,Q)}((M[i] * \sum_{k=1}^{N(nodes)} N(OP_i,k)) \leq M_{max}[i])$$
$$\forall_{i=1}^{N(OP,Q)}((M[i] * \sum_{k=1}^{N(nodes)} N(OP_i,k)) \geq M_{min}[i])$$

Deriving the shelf constraints is a bit more involved as not all operators may execute on all nodes. $N(OP_{S_j,i},k)$ denotes the number of clones of the ith operator in the jth shelf which are executing at node k. The shelf constraints ensure that the sum total of the memory used by all the operator clones executing at a node does not exceed the amount of memory available to the query at that node. This gives rise to $N(S,Q) * N(nodes)$ shelf constraints:

$$\forall_{j=1}^{N(S,Q)}\forall_{k=1}^{N(nodes)}((\sum_{i=1}^{N(OP,S_j)} N(OP_{S_j,i},k) * M[OP_{S_j,i}]) \leq M_{sys}[k])$$

The new objective function is given by

$$B(Q,M) = \sum_{i=1}^{N(OP,Q)} B(OP_i) * M[i] * (\sum_{k=1}^{N(nodes)} N(OP_i,k))$$

As the objective function and all the constraints remain linear in $M[i]$, we can still use the LINEAR algorithm to find out the optimal query memory allocation for a query running on a multi–node parallel database system.

7 Conclusion

To our knowledge, the problem of allocating memory to multiple operations in a complex decision support query has not been addressed previously. Past research in this area has either dealt only with single join queries or has assumed the availability of sufficient memory to completely satisfy the demands of all operators in the query. We believe that even in a moderately loaded multi-user environment it is more realistic to assume that queries which have several memory intensive operations might generally be given memory less than their maximum requirement.

In this paper we have described memory management schemes that can cope with such a situation by taking advantage of the malleability of hash join and hash aggregation operations. We have presented four algorithms that can distribute available memory among operators in a query in a manner that satisfies all operator requirements and concurrent schedulability constraints. One particular algorithm (LINEAR), based on linear programming, is in fact guaranteed to produce an *optimal memory allocation* in terms of the best query response time. However another heuristic based algorithm (SMALL) is much easier to implement and usually generates memory allocations that perform reasonably well.

We have demonstrated experimental results corroborating our analytical findings and we have also shown how these basic memory allocation schemes can be adapted to shared-nothing parallel database systems. We believe that the implementation of effective memory allocation algorithms will prove to be a major performance advantage to any database system running complex, decision support queries.

References

[1] S. Chaudhuri and U. Dayal. "An Overview of Data Warehousing and OLAP Technology". *SIGMOD Record*, **26**(1), 1997.

[2] M.S.Chen, M.L. Lo, P.S. Yu, H.C. Young. "Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins". *Proc. of VLDB Conf.*, 1992.

[3] D.L. Davison and G. Graefe. "Memory Contention Responsive Hash Joins". *Proc. of VLDB Conf.*, 1994.

[4] M.N. Garofalakis and Y.E. Ioannidis. "Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources". *Proc. of VLDB Conf.* 1997.

[5] W. Hasan, D. Florescu and P. Valduriez. "Open Issues in Parallel Query Optimization". *ACM SIGMOD Record*, **25**(3) 1996.

[6] M. Mehta and D.J. DeWitt. "Dynamic Memory Allocation for Multiple Query Workloads". *Proc. of VLDB Conf.*, 1993.

[7] M. Mehta, D. J. DeWitt. "Managing Intra-operator Parallelism in Parallel Database Systems". *Proc. of VLDB Conf.*, 1995.

[8] H. Pang, M.J. Carey and M. Livny. "Managing Memory for Real-Time Queries". *Proc. of ACM SIGMOD Conf.*, 1994.

[9] J.M. Patel et al. "Building A Scalable GeoSpatial Database System: Technology, Implementation, and Evaluation". *Proc. of ACM SIGMOD Conf.*, 1997.

[10] W.H. Press et al. "Numerical Recipes in C". *Cambridge Univ. Press*, 1992.

[11] R. Gerber, *Informix Corporation*. Private Communication.

[12] E. Rahm and R. Marek. "Dynamic Multi-Resource Load Balancing in Parallel Database Systems". *Proc. of VLDB Conf.*, 1995.

[13] D.A. Schneider and D.J. DeWitt. "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines". *Proc. of VLDB Conf.*, 1990.

[14] P. G. Selinger et al. "Access Path Selection in a Relational Database Management System". *Proc. of ACM SIGMOD Conf.*, 1979.

[15] J. Srivastava and G. Elsesser. "Optimizing Multi-Join Queries in Parallel Relational Databases". *Proc. of Conf. on PDIS.* Jan 1993.

[16] The Transaction Processing Council. "TPC-D Benchmark Specification Rev.1.3.1". *http://www.tpc.org/dspec.html*, 1998.

[17] The Transaction Processing Council. "TPC-D Individual Results (Executive Summaries)". *http://www.tpc.org/execsum_TPCD.html*, 1998.

[18] P.S. Yu and D.W. Cornell. "Buffer Management Based on Return on Consumption in a Multi-Query Environment". *VLDB Journal*, **2**(1), Jan 1993.

[19] H. Zeller and J. Gray. "An Adaptive Hash Join Algorithm for Multiuser Environments". *Proc. of VLDB Conf.*, 1990.