VLDB Journal, 2, 1-37 (1993) Hector Garcia- Molina, Editor

Buffer Management Based on Return on Consumption In a Multi-Query Environment

Philip S. Yu and Douglas W. Cornell

Received November 26, 1990; revised version received January 5, 1992; accepted July 6, 1992.

Abstract. In a multi-query environment, the marginal utilities of allocating additional buffer to the various queries can be vastly different. The conventional approach examines each query in isolation to determine the optimal access plan and the corresponding locality set. This can lead to performance that is far from optimal. As each query can have different access plans with dissimilar locality sets and sensitivities to memory requirement, we employ the concepts of memory consumption and return on consumption (ROC) as the basis for memory allocations. Memory consumption of a query is its space-time product, while ROC is a measure of the effectiveness of response-time reduction through additional memory consumption. A global optimization strategy using simulated annealing is developed, which minimizes the average response over all queries under the constraint that the total memory consumption rate has to be less than the buffer size. It selects the optimal join method and memory allocation for all query types simultaneously. By analyzing the way the optimal strategy makes memory allocations, a heuristic threshold strategy is then proposed. The threshold strategy is based on the concept of ROC. As the memory consumption rate by all queries is limited by the buffer size, the strategy tries to allocate the memory so as to make sure that a certain level of ROC is achieved. A simulation model is developed to demonstrate that the heuristic strategy yields performance that is very close to the optimal strategy and is far superior to the conventional allocation strategy.

Key Words. Buffer management, query optimization, simulated annealing, join methods, queueing model, simulation.

1. Introduction

Database systems have generally relied on memory buffers to reduce disk accesses. Even with the trend of ever-increasing memory size, the memory buffer usually can

Philip Yu, Ph.D., is Manager, Architecture Analysis and Design Group, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; Douglas W. Cornell, Ph.D., is Principal Software Engineer, Digital Equipment Corp., Littleton, MA.

not accommodate all the databases in the system, and some memory management strategy is needed to make the best use of the memory space. The traditional approach to memory management in a virtual memory environment often uses a least recently used (LRU) replacement policy, which replaces the least recently used page with a newly referenced page to capture temporal locality. For a network or hierarchical database system, reference strings tend to be unpredictable except for batch processing. A study of network databases can be found in (Effelberg and Loomis, 1984). These types of systems seem to fit reasonably well with the working set model. However, queries to relational databases (Codd, 1970) imply a lot of information on data references. The query optimizer analyzes each query and generates an access plan which contains detailed information on how each relation is accessed. Although some variant of the LRU policy often is used for buffer management, it is not considered to be well suited for the reference patterns of relational databases (Stonebreaker, 1981).

In a relational database environment, queries that do not involve join operations have only a small memory requirement and we therefore concentrate our attention on how join operations are affected by memory availability. Three methods have commonly been used for performing the join: hash join, sort-merge join, and nested-loop join. Each of these join methods can operate under different memory allocations with dissimilar performances. There is some minimum amount of working storage required for each join method. The working storage includes the I/O buffering for each joining relation and the additional storage, (e.g., for the hash table under the hash-join method or the sort buffer, like the tournament tree [Knuth, 1975]), under the sort-merge join method. Beyond the minimum requirement, more storage may be allocated to speed up the query. In hash joins, the amount of storage allocated for the hash table determines the proportion of the tuples needed to be read more than once (Shapiro, 1986). In sort-merge joins, the amount of storage available for the sort buffer determines the number of sorted runs generated after the sort phase.

Previous research directed toward using information available about reference patterns for buffer management has been described (Sacco and Schkolnick, 1982, 1986; Chou and DeWitt, 1985). In Sacco and Schkolnick (1982, 1986) a hot-set model is proposed for buffer allocation. The basic idea is to determine a hot set for every query and allocate sufficient buffer space to cover the maximum hot set that will fit the buffer constraint before executing a query. It can lead to substantial performance improvement over the LRU strategy. However, this is a local optimization for each query to provide it with sufficient buffer space to minimize disk I/O accesses. The potential buffer contention among queries is not

addressed in the buffer allocation strategy. As pointed out in Sacco and Schkolnick (1986), straightforward implementation of this idea can lead to problems such as infinite waits, long queries blocking short queries, etc. Some ad hoc techniques to relieve these problems are also suggested. Active instances (due to references from different queries) of a file are given different buffer pools and are managed by different replacement disciplines (Chou and DeWitt, 1985). A DBMIN algorithm proposed for estimating the buffer allocation and replacement discipline for each file instance of a query is described. DBMIN is based on a query locality set model to capture relational query behavior. All these works investigate the "right" buffer allocation for a given query plan or access path selection without considering the effect of other queries.

In the presence of multiple queries simultaneously under execution, how much memory to be allocated to each query and which join methods to be employed can not easily be determined. For each query type, different access plans show dissimilar sensitivities to memory allocation and thus have distinct memory requirements. The best access plan for a given memory allocation is not necessarily the best plan for another memory allocation. Even for a pre-selected join method, like hash join or sort-merge join, the appropriate working storage to be allocated in a multi-query environment is far from clear. The return on memory allocation to dissimilar queries can be vastly different. In one extreme, one can allocate just the minimum working storage requirement to each query if memory is the highly contended resource. The other extreme is to allocate the maximum requirement to each query—to accommodate the whole relation in the hash table under hash join or to sort the whole relation in one sorted run under sort-merge join—if there is ample memory. However, in most cases, one needs to allocate something in between the two extreme points, and there is a big gap between the minimum and the maximum requirement. (The minimum, as we shall see later, is roughly the square root of the maximum.) This is especially the case when the relation size is larger than the buffer size. The concept of a hot set or query locality set does not provide a meaningful indication of the appropriate amount of memory allocation in this case. Under the hot set strategy the minimum requirement is allocated, whereas under the query locality set approach the maximum allowed for any query is allocated similar to a fixed allocation scheme. Cornell and Yu (1989) proposed an integrated strategy based on an integer programming approach to allocate storage and make access plan selection when arrival rates of all query types are known. The allocation strategy specifically considers which relations should be kept in the memory during the join. In Ng et al. (1991), generalizing DBMIN, a class of algorithm based on marginal gains is proposed and studied. It shows that suboptimal allocations, when handled properly,

can lower the waiting time for buffer and improve overall system performance.

Here we introduce the concept of return on consumption (ROC) to guide the memory allocation. The memory consumption of each query is defined to be its space time product. ROC is introduced as a measure on the effectiveness of additional memory consumption on response time improvement. Note that the cost of allocating additional memory depends not only on the amount of memory allocation but also the length of time the memory is in use. That is to say, memory consumption is a better measure of the cost of additional memory allocation. Consider an example. Assume that query A has a 10-sec response time at a 50-page memory allocation and query B has a 20-sec response time at a 40-page allocation. Let's assume that an additional 100-page allocation will reduce the response time of either query by 5 sec. Although the benefit is the same, the cost is very different. Query B will have an increase in memory consumption of 1300 page-sec (= 15 *sec* $\times$ 140 *pages* - 20 *sec* $\times$ 40 *pages*) while query A only has an increase of 250 page-sec. Even if query B achieves a 10-sec response time reduction with the additional 100 page allocation, the increase in memory consumption will be 600 page-sec, which is still more than twice that of query A. As we shall see later, ROC can indeed provide the appropriate metrics to decide the memory allocations among queries.

Practical memory management schemes need to be simple and should not rely on perfect knowledge of the workload. However, it is generally hard to evaluate the optimality of a heuristic strategy, especially in a multi-query environment. Here we first develop an approach to find the global optimal memory allocation strategy assuming perfect knowledge on the workload mix. This provides an upper bound on the achievable performance based on which more realizable strategies can be compared. A heuristic strategy is then devised without relying on this assumption and its performance is found to be reasonably close to the optimal solution. Specifically, a global optimization strategy, based on simulated annealing (Kirkpatrick et al., 1983, Kirkpatrick and Toulouse, 1985) is developed to select optimal join methods and memory allocation. It handles the case when relation size is larger than the buffer size and decides the fraction of the relation to be kept in the buffer. The objective function is the average response time over all query types and the major constraint is that the memory consumption rate over all query types needs to be less than or equal to the total buffer size. By analyzing the solution from the optimal strategy, a heuristic strategy is proposed based on ROC.

In Section 2 we examine the ROC under different join methods. We describe the global optimization procedure for buffer management in Section 3 and the heuristic threshold strategy in Section 4. In Section 5, a performance comparison of the different strategies is presented. Concluding remarks are given in Section 6.

2. Return on Memory Consumption

We now take a closer look at the concepts of memory consumption and ROC. Formally, memory consumption of a query is defined to be the integral of the instantaneous memory allocation, $F(t)$, over its in-memory time, T, i.e., $\int_0^T F(t)dt$. A complex query can be viewed as a sequence of steps where each step is either a join or some retrieve or projection operation. If the memory allocation stays the same during the in-memory time of a query step, the memory consumption of a query step can be expressed as the in-memory time of the query step multiplied by its memory allocation. Note that the in-memory time here is from the time the query step is initiated until it is completed. It does not include the time waiting for memory to become available. For the rest of this article, we assume that the memory allocation is by query step and each query consists of a single step to simplify the discussion. Generalization is straightforward. Clearly, the memory consumption per unit time over all queries can not be larger than the total buffer size. Thus the memory allocation problem can be viewed as an optimization problem to achieve the most improvement in response time under the constraint that the total memory consumption rate has to be less than the buffer size.

ROC is introduced as a measure on the effectiveness of additional memory consumption on response time improvement. Let F_{min} be the allowable memory allocation that achieves the minimum memory consumption to execute a given query access plan. (If there are multiple memory allocations that can achieve the minimum consumption, F_{min} will be set to the largest allocation, as it provides the minimum response time.) Memory allocation $< F_{min}$ is generally not meaningful as it takes longer to execute and causes more memory consumption. For memory allocation $F > F_{min}$, the response time or the in-memory time, $T(F)$, cannot be larger than $T(F_{min})$, assuming the additional memory is employed in a meaningful way. The ROC at a memory allocation, $F > F_{min}$, is the reduction in in-memory time, $T(F) - T(F_{min})$, divided by the additional memory consumption relative to the point of minimum memory consumption:

$$ROC(F) = \frac{T(F_{min}) - T(F)}{T(F)F - T(F_{min})F_{min}}.$$

Note that ROC is a more meaningful metrics than the return on memory allocation (ROA):

$$ROA(F) = \frac{T(F_{min}) - T(F)}{F - F_{min}},$$

which can be viewed as a surrogate for the negative of the derivative of $T(F)$ in the discrete variable F. This is due to the fact that the cost of allocating additional

memory depends not only on the amount of memory allocation but also the length of time the memory is in use. Also note that $T(F)$ is the in-memory time, and does not include the time the query must wait for its memory allocation to become available. Hence, ROC does not depend on the level of memory contention in the system. In the example given in the previous section, both queries A and B have the same ROA, but query A has a much larger ROC, assuming both queries originally are operating at the minimum consumption point. Furthermore, even after the ROA of query B is doubled, query A still has a larger ROC.

In a relational database environment, queries that do not involve join operations have only a small memory requirement and we therefore concentrate our attention on how join operations are affected by memory availability. Three methods have commonly been used for performing the join: hash join, sort-merge join, and nested-loop join. Each of these join methods can operate under different memory allocations with dissimilar performance. We examine here the effect of memory allocation on query response time and memory consumption for individual two-way join queries running alone in the system. The sensitivity of ROC to different memory allocations is then considered. The effect of resource contention among concurrently executing queries is treated in Section 3, where a global optimization strategy based on simulated annealing is developed. The findings in this section provide a guide to reduce the search space for the global optimization procedure. There are many different ways of implementing any of these join methods. We would only pick a simple implementation for each join method to illustrate the concepts of memory consumption and ROC.[1] For an alternative implementation, we can similarly derive the response time curve (vs memory allocation) and then the corresponding set of memory consumption and ROC curves. Even for more complex multi-way joins, which often involve pipelining into succeeding joins, as long as we can estimate the response time curve, the corresponding set of memory consumption and ROC curves follows directly. The buffer management approach introduced in this paper only relies on these concepts and does not depend on any particular implementation of the join method, pathlength parameters, or formula to estimate CPU and I/O overhead.

2.1 Hash Join

Let R and S be the two relations to be joined, with R being the smaller relation.

1. The simple implementations are chosen to show different ROC behaviors. The intent is not to use the most efficient implementations to compare the different join methods.

When R is smaller than the available memory M, the hash join algorithm works as follows: the join attributes of all the tuples from R are first hashed and a hash table is built in memory. Then relation S is scanned sequentially. For each tuple in S, the join attribute is hashed and used to probe R's hash table in memory. When a match is found the corresponding tuples from the two relations are concatenated and added to the result relation.

Three extensions to the above algorithm for situations where R exceeds the size of available memory have been presented (Dewitt et al., 1984; Shapiro, 1986). Among the three, the hybrid hash join algorithm has been shown to be the most efficient algorithm. We assume the hybrid hash join in this article.

Hybrid hash join consists of three phases. In phase one, relation R is read from disks and is hashed into multiple partitions where only the hash table of the first partition is kept in memory. The other partitions are stored on disks. The number of partitions is chosen such that the hash table for each partition will individually fit in the memory. In the second phase, relation S is read from disks and is similarly hashed into multiple partitions based on the same hash function. Tuples from its first partition are joined directly with those of relation R residing in the memory. The other partitions are stored on disks. In the third phase the remaining partitions are joined. For each pair of corresponding partitions, tuples from relation R are first hashed to build a hash table in memory and then tuples from relation S are used to probe the hash table to find a match.

We next examine the minimum memory allocation required to make relation R partitionable so that each of its partitions can fit in memory. Let M be the size of the available database buffer. For relation R, let $|R|$ be the number of pages in the relation. Let δ be the expansion factor of the hash table relative to the partition size. The minimum value of M should be greater than $\sqrt{\delta|R|}$ where relation R is the smaller of the two joining relations (Dewitt et al., 1984). For memory allocation beyond the minimum, the size of R_0 will be increased, thus reducing the portion of relations needed to be read in twice. Formulas to evaluate the performance sensitivity to memory allocation are given in the Appendix.

Assume the pathlength to perform an I/O operation is I_{io}, the pathlength to either extract each tuple from the input buffer or move it to the output buffer is I_{move}, the pathlength to apply a hash function to the join attribute value is I_{hash}, the pathlength to search for a match or empty slot in the hash table is I_{search}, and the pathlength to join a pair of tuples is I_{join}. I_{search} may involve several comparisons or hash collisions. The number of comparisons is affected by the the expansion factor, δ. By maintaining a fixed expansion factor, δ, the memory allocations will not affect I_{search} and only change the portion of data to be read in

Figure 2.1 Single-query response time for Hash Join

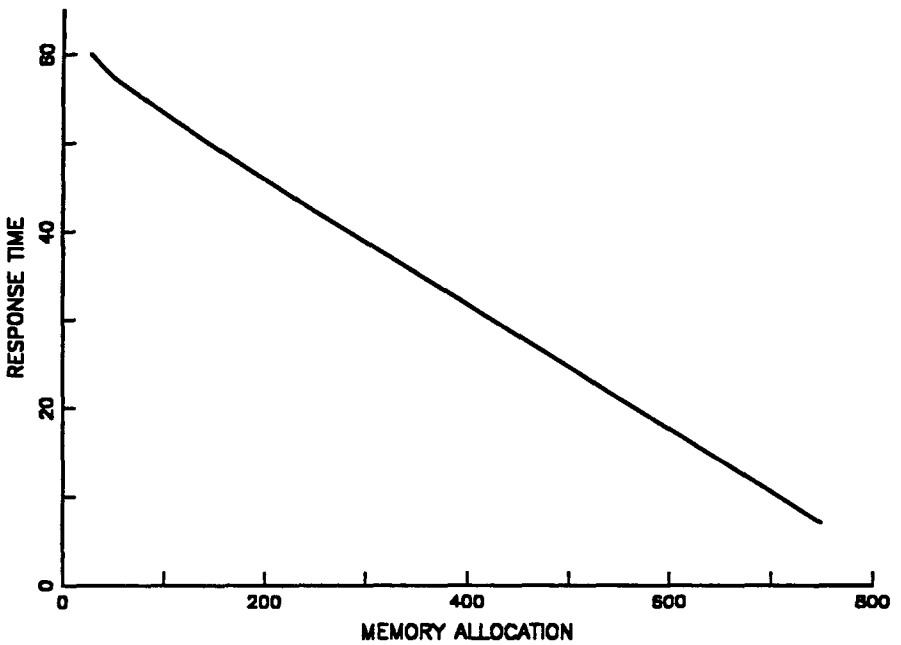


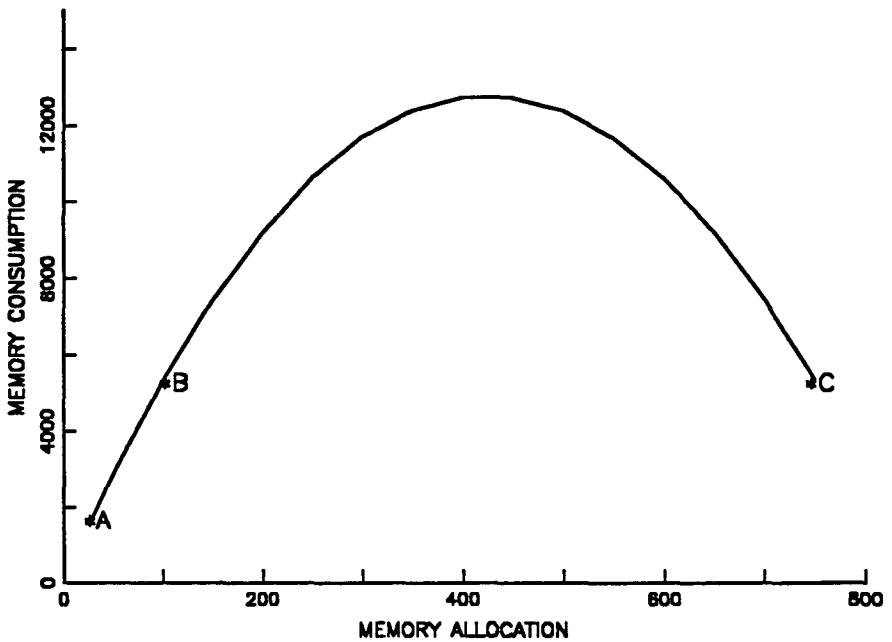
Table 2.1 System configuration

Processor speed	15 MIPS
No. of disks	10
Disk I/O time	30 millisec

twice. The I/O and CPU requirements can be derived in terms of these parameters (Lakshmi, 1989) (see Appendix). The single query response time is simply the sum of the CPU time and the I/O time.

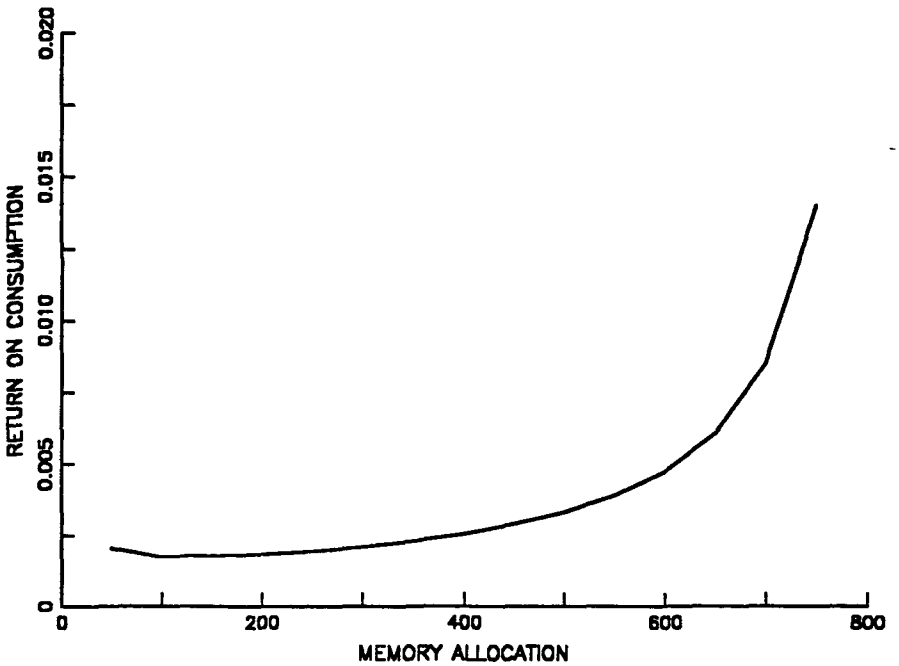
Consider an example of a query joining two relations of 500 and 1,000 pages with 10,000 and 20,000 tuples, respectively. The join selectivity, which is defined to be the cardinality of the join result divided by the product of the cardinalities of the two joining relations, is assumed to be 0.001. Throughout this article, we assume a system configuration given in Table 2.1. Furthermore, the following pathlength parameters are assumed: $I_{io} = 3000, I_{move} = 500, I_{hash} = 100, I_{search} = 1000,$ and $I_{join} = 100$. A prefetch blocking factor of 10 pages also is assumed to scan through each relation when hash join is considered. We use an expansion factor δ of

Figure 2.2 Memory consumption for Hash Join



1.5. A plot of the single query response time vs memory allocation is shown in Figure 2.1. For hash joins, increasing the memory allocation has almost a linear effect on reducing the query response time which is also evident from the analysis in the Appendix. The corresponding plot of memory consumption vs memory allocation is shown in Figure 2.2. The shape of the curve is concave. Point A corresponds to the point of minimum allocation whereas point C corresponds to the point of maximum allocation where the hash table can accommodate the whole relation (i.e., the hash join can be done in one pass through the relation). Assume point B has the same memory consumption as point C (i.e., a point with much less memory allocation and longer response time), but with the same product of the two quantities as point C. If C represents a feasible allocation, allocations between B and C are not meaningful since they result in more buffer consumption and longer response time than point C. Points between A and B correspond to the situation operating with lower memory consumption but larger response time. We may be forced to operate in this region, if the system is memory limited. Thus the curve

Figure 2.3 Return on consumption for Hash Join



implies the only sensible allocation would be either the maximum allocation, or some allocation between A and B. If available buffer is less than the relation size, let C' represent the point corresponding to the maximum allowed memory allocation, the entire buffer. If C' is on the right of the maximum memory consumption point, there is another point B' at the left of the maximum memory consumption point which has the same memory consumption as C'. A similar argument can be made that the only sensible allocations would be either the maximum allocation at C', or some allocation between A and B'. If C' is on the left of the maximum memory consumption point, the only sensible allocations would be some allocations between A and C'. Figure 2.3 shows the ROC versus memory allocation. Although the ROC is mostly increasing (except in the beginning), the curve does not take off until it nears the maximum allocation. This seems to imply that in a multiple query environment, the strategy to get maximum return from the memory allocation is not to equally divide the buffer among all queries, but to give more memory to a few smaller queries so they can operate at the point of high return and to give the

big queries close to their minimum requirement. Here small queries mean queries joining relations of smaller sizes. In the next section, we shall see this is exactly what happens under the global optimization strategy.

2.2. Sort-Merge Join

The sort-merge join can be viewed as consisting of three phases. First, tuples from each relation are scanned and sorted runs are produced. This is referred to as the scan phase. Next, the sorted runs are merged to produce a single sorted run for each relation. This is referred to as the merge phase. The third phase is the actual join phase, where the two sorted relations are joined by merging the matching tuples on the join attribute. One variation of the above scheme is to combine phase 2 and phase 3 together when there is sufficient memory. We ignore this optimization as it does not affect the buffer management methodology.

The scan phase can be implemented using a tournament tree sort (Knuth, 1975). In this method, tuples are placed into the leaves of the tournament tree. Values on the join attribute of tuples at the same level of the tree are compared and the tuple with the smallest value is moved to a level higher in the tree. Once the tree is full, the tuple at the root is output and a new tuple is inserted into the tree. On the average, the sorted runs of tuples are twice as long as the number of tuples that can fit into the tournament tree.

The performance of the sort-merge join depends critically on the number of passes required through each relation. Each additional pass means reading in and writing out the relation one more time. There are three critical memory allocations which affects the number of passes. The first one occurs at the point that the merge phase can be accomplished in one pass. This memory requirement would be taken as the minimum required allocation for sort-merge join as failing to do this would result in additional passes through the relation, hence a substantial increase in I/Os. Let ϕ be the expansion factor of memory requirement due to pointers in the nodes of the tournament tree. Assume that S is the larger of the two joining relations. $|S|$ is defined in the same way as $|R|$. The minimum allocation of M can be shown to be $> \sqrt{\phi|S|}$ (DeWitt et al., 1984). In comparing the sort-merge join with hash join, we note that the minimum required allocation for the sort-merge join is proportional to the square root of the larger relation whereas the minimum required allocation for the hash join is proportional to the square root of the smaller relation.

The second critical memory allocation occurs at the point when the tournament tree is large enough so that the smaller of the two joining relations can result in one

sorted run, thus eliminating the merge phase for that relation. This saves one pass over the smaller relation by eliminating the necessity of reading in the relation for the merge of runs and rewriting to disk. The size of the tournament tree would be roughly equal to half of the smaller relation size. A third critical memory allocation occurs at the point when the tournament tree is large enough so that the larger of the two joining relations can also result in one sorted run.

Next we consider the impact of additional memory in between these critical points. Consider the case when the number of tuples is a power of 2. Doubling the memory space for the tournament tree increases the number of comparisons by 1 during the scan phase and reduces the number of sorted runs by half. At the merge phase, as the number of runs is reduced by half, the level of comparisons can be reduced by one. Hence, the CPU time is not sensitive to the memory allocations as long as the number of passes through each relation remains the same. For the general case, in which the I/O time is a substantial component of the response time and the instruction overhead for comparison per level of tournament tree is small, even if the additional memory allocations results in one fewer or one more comparisons, it would hardly make a difference in the overall response time as long as the number of passes through each relation remains the same. Detailed analysis of the CPU and I/O requirement can be found in the Appendix. The CPU requirement can be expressed in terms of the various pathlength parameters, like I_{io}, I_{move}, and I_{comp}. The last parameter is the pathlength to do a comparison per level on the tournament tree.

Next we consider an example, again assuming a query joining two relations of 500 and 1,000 pages with 10,000 and 20,000 tuples, respectively, and a join selectivity of 0.001. A blocking factor of 10 is assumed for the I/O operation during the sort phase and join phase. For the merge phase, the potential large number of sorted runs can make it prohibitive to merge in one pass if each run requires a large input buffer, hence no prefetch is assumed for the input operation. A plot of the estimated single query response time for performing the join is plotted versus memory allocation in Figure 2.4 for pathlength parameter values of $I_{io} = 3000, I_{comp} = 100$, and $I_{move} = 500$. We assume in this example the worst case scenario: that the number of nodes needed in the tournament tree to produce one sorted run of either relation is equal to the cardinality of the relation. The response time curve shows two drops at the points where each of the relation can be sorted in one run, respectively. In between these two points and the point of minimum memory allocation, the additional memory allocations does not lead to any noticeable improvement in response time. The corresponding plot of memory consumption versus memory allocation is shown in Figure 2.5. The

Figure 2.4 Single query response time for Sort-Merge Join

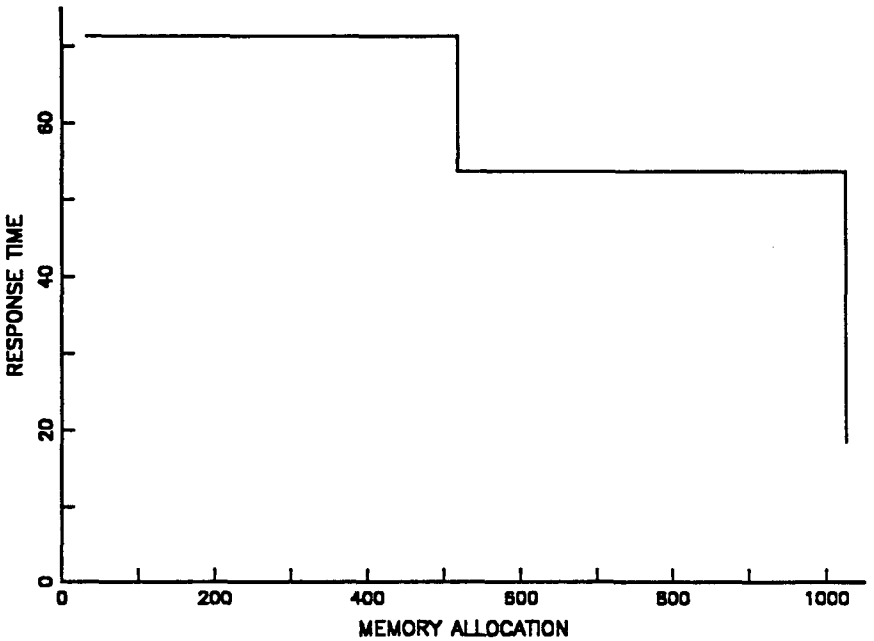
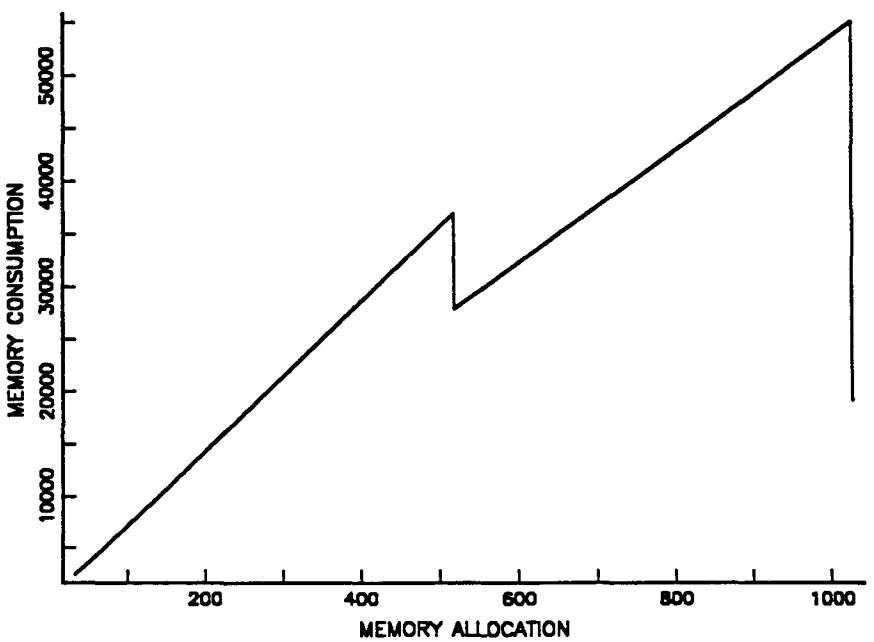


Figure 2.5 Memory consumption for Sort-Merge Join

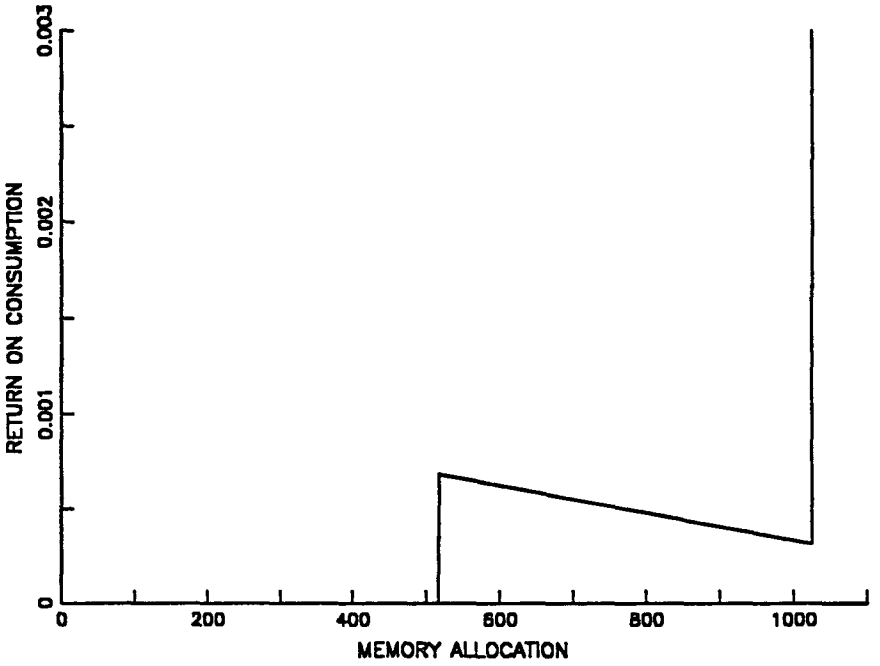


memory consumption achieves its local minimum at these three points. In between there is only increase in consumption with no improvement in response time. Figure 2.6 shows the ROC versus memory allocation. Again ROC reaches its high points at the two points where the merge phase can be eliminated. In between ROC is either zero (i.e., no return), or decreasing. It is clear from these figures that the memory allocation should be either the minimum memory needed to do the merge in one pass or should be the memory allocations to eliminate the merge phase of the sorted runs. These allocations actually correspond to the hot sets of Sacco and Schkolnick (1986). The difference is that those authors addressed only the single query environment so the maximum hot set that would fit into the whole buffer is picked. This will be useful guidance in pruning the search space for the global optimization in the next section. Furthermore, we note that an alternative implementation to take better advantage of the buffer allocation in-between the critical sizes is to use the additional memory to store runs between phases, thus saving I/O costs (Shapiro, 1986).

2.3. Nested-Loop Join

One implementation of a nested-loop join is a method based on table scan where one scan of an "inner" relation takes place for each tuple present in the "outer" relation (Selinger et al., 1979). The minimum memory allocation is two blocks: one for the inner relation and the other for the outer relation. The outer relation only needs to be read in once, while the inner relation will be read in as many times as the number of tuples or blocks in the outer relation. If the inner relation can be kept in memory, both relations only need to be read in once. This amount of memory allocation to keep the inner relation in memory is referred to as the hot set size (Sacco and Schkolnick, 1986). If the memory allocation is in between the minimum amount and the hot set size, the portion of the inner relation that will be read in multiple times depends upon the memory management policy. Let us consider two different memory management policies: MRU and LRU. The analysis of the I/O and CPU requirement is included in the Appendix. If an MRU type policy is used to manage the memory, the additional memory allocation will have a linear effect in reducing the portion of the inner relation that needs to be read in repeatedly (Chou and DeWitt, 1985). The response time curve versus memory allocation has a shape similar to Figure 2.1. Furthermore, the shapes of the memory consumption curve and ROC curve would be similar to those of Figures 2.2 and 2.3, respectively. If an LRU type policy is adopted, additional memory allocation beyond the minimum allocation is useless. It behaves no better than under the minimum allocation. The

Figure 2.6 Return on Consumption for Sort-Merge Join



response time is a step function similar to that in Figure 2.4, ignoring the portion beyond the first drop. The memory consumption curve and ROC curve would be similar in shape to those of Figures 2.5 and 2.6, respectively. Thus all comments in the previous subsections on memory allocations in the multiple query environment again apply here.

3. Global Optimization Strategy

The conventional approach to query optimization is to examine each query in isolation and select the access plan with the minimal cost based on some predefined cost function of I/O and CPU requirements to execute the query (Selinger et al., 1979). The impact of memory management generally is not captured in the cost function. Furthermore, the value of the cost function does not reflect the potential effect of other transactions concurrently under execution.

As we have seen in the previous section, for a given query, different access plans show different sensitivities to memory allocation. The best access plan for a given

memory allocation is not necessarily the best plan for another memory allocation. In this section, a global optimization strategy is developed which considers not only query access plan selection but also the optimal buffer allocation in a multi-query environment. It takes all query types into consideration assuming the arrival rate of each type is known. (The assumption on query arrival rate is relaxed later on when the heuristic threshold strategy is considered in Section 4.) This type of global optimization has also been considered for the case where the relation size is less than the buffer size (Cornell and Yu, 1989). They proposed a (0,1) integer programming approach to decide whether each relation referenced by a query should be kept in memory in its entirety or not. It is shown that the system performance can be drastically improved under the global optimization approach. Here, we consider the situation that the relation size can be larger than the buffer size. A generalized approach based on simulated annealing is adopted to allow for any fraction of a relation to be kept in memory based on memory availability and query mix. The result of the global optimization will serve as the basis to understand memory allocation and derive heuristic strategy in Section 4.

3.1 Optimization Problem

Consider a set of join queries $Q_i, i = 1, \cdots, N_Q$, and relations $R_k, k = 1, \cdots, N_R$. For each query type Q_i, there are the three join methods and each can operate under different memory allocations with different sensitivities to performance. Decision variables X_{ij} are used to specify the join method currently under consideration for Q_i with $j = 1, 2, 3$ referring to hash join, sort-merge join, and nested-loop join, respectively. $X_{ij} = 1$ if the j-th join method is adopted for Q_i, otherwise $X_{ij} = 0$. Let F_i be the amount of memory allocation for query i. For each strategy η, there is an associated (X_{ij}, F_i) for Q_i. Furthermore, for each Q_i, let $D_{ij}(F_i)$ be the number of pages to be read from disks if the j-th join method is used with a memory allocation of F_i. Similarly, $U_{ij}(F_i)$ can be defined for the query-processing pathlength at the CPU. Formulas to estimate $D_{ij}(F_i)$ and $U_{ij}(F_i)$ are given in the Appendix, where $U_{ij}(F_i)$ is expressed in terms of the various pathlength parameters like I_{io}, I_{move}, I_{comp}, and I_{hash}. Define λ_i to be the arrival frequency for Q_i.

Now we formulate the optimization problem. The objective function is the average response time over all query types. Let $RT_{ij}(F_i)$ be the average response time of Q_i under the j-th join method with a memory allocation of F_i. Thus,

$$\text{objective function} = \sum_j \sum_i \lambda_i RT_{ij}(F_i) X_{ij}.$$

The average response time in a multi-query environment is determined using an open queueing network model. Assume the system consists of a single CPU with speed $MIPS$ and multiple (N_D) disks. Each database is assumed to be partitioned uniformly across the N_D disks, based on its primary key. For a given strategy η, the response time of each query can be calculated as its resource requirement on the CPU and disks can be predicted given the buffer allocation. The total CPU processing cost is then

$$U_{CPU} = \sum_j \sum_i \lambda_i U_{ij}(F_i) X_{ij}.$$

Let $T_{I/O}$ be the disk service time to perform an I/O operation. The utilization of each disk assuming the load is spread uniformly across all disks for each relation can be shown to be

$$\rho_{I/O} = \frac{T_{I/O}}{N_D} \sum_j \sum_i \lambda_i D_{ij}(F_i) X_{ij}.$$

The average response time of Q_i is

$$RT_{ij}(F_i) = \frac{U_{ij}(F_i)}{MIPS - U_{CPU}} + \frac{D_{ij}(F_i) T_{I/O}}{1 - \rho_{I/O}},$$

where the first component is the sum of the service time and waiting time at the CPU and the second component is the sum of those times at the disks.

Constraints are included to guarantee that exactly one join method is adopted for each query,

$$\sum_j X_{ij} = 1, \quad j = 1, \cdots, 3.$$

Additional constraints are added to prevent the buffer from being overcommitted. First of all, no query can get a memory allocation more than the total buffer size. Let B be the number of memory pages available.

$$F_i \leq \alpha B, \qquad \text{for each } i.$$

The α is chosen so that a reasonable multiprogramming level can be maintained. Furthermore, the memory consumption rate of each query type is estimated as the product of memory allocation and response time $\times$ the arrival frequency. In order to provide a margin of safety, thus accommodating the fluctuations in query workload, the memory consumption rate needs to be less than some fraction, β, of the total buffer.

$$\sum_j \sum_i \lambda_i F_i RT_{ij}(F_i) X_{ij} \leq \beta B.$$

An implicit assumption of the optimizing procedure is that α and β are chosen such that if the memory allocation of any single query is less than some fraction α of the buffer size and the total memory consumption rate is less than some fraction β of the buffer size, then memory is not the bottleneck.

3.2 Solution Technique

To solve for the above optimization problem, we use the method of simulated annealing (Kirkpatrick et al., 1983; Kirkpatrick and Toulouse, 1985). It was invented as an optimization analogy to the statistical mechanics associated with annealing solids. This approach has been applied successfully to problems having several thousand variables (Kirkpatrick et al., 1983) and a variety of applications, including query optimization (Ioannidis and Wong, 1987; Ioannidis and Kang, 1990, 1991; Swami and Gupta, 1988; Swami, 1989) and file assignment (Wolf et al., 1989). We outline the general simulated annealing algorithm as follows (Kirkpatrick et al., 1983).

Pick initial feasible solution S
Pick initial temperature T
Do while (not frozen):
  Do while (not in equilibrium):
    Pick random nearby feasible solution S'
    Let Δ be equal to the difference in objective function values
      between the two allocations S and S'.
    If $\Delta < 0$ then set $S = S'$
    Else set $S = S'$ with probability $e^{-\Delta/T}$
  End.
  Reduce temperature T.
End.
Final solution is S.

To determine whether a strategy, $\{(X_{ij}, F_i), i = 1, ..., N_Q\}$, is feasible, we need to check if it satisfies all the constraints. The memory consumptions of all query types need to be calculated and the sum of memory consumptions over all query types must be less than β times the buffer size. The initial temperature T is chosen to be $T_0(= 10)$. At this temperature, nearly all the nearby feasible solutions will be successes, since $e^{-\Delta/T}$ will be close to one. As T decreases, successes for poorer solutions will become scarcer. The average response time under the current strategy is then estimated and compared to that of the previous cycle of

calculation. In the following examples, to reduce the temperature, we replace T with $\psi(= 0.9)T$. Furthermore, equilibrium is defined to be the point that either $k_S(= 10)(N_Q + N_R)$ successes have occurred or $k_F(= 50)(N_Q + N_R)$ failures have occurred. Freezing is defined to be the point where we continue to cool until $k_L(= 3)$ losing temperatures have occurred in a row. The parameters (namely, T_0, ψ, k_S, k_F, and k_L) chosen for the particular cooling schedule were based on experiments to achieve a balance between efficiency and accuracy. This cooling schedule is similar to the one of Kirkpatrick et al. (1983). More elaborate cooling schedules can be found (Van Laarhoven, 1987), but their utility in this particular problem was found to be marginal.

At each cycle of the calculation, a random nearby solution is chosen as follows: A new candidate query is picked by random number generation to have its join strategy altered either by using a different join method or by different memory allocation. From the analysis in the previous section we know that for each join method, only certain memory allocations are meaningful. For example, if the join method is sort-merge, a meaningful memory allocation must correspond to one of the three local minima of Figure 2.4, and if hash join, the meaningful allocation is the allocation's correspondence to point C and the region between points A and B. The memory allocation is adjusted by some fractional amount depending on the value returned by the random number generator. If the increase in memory allocation for the candidate query causes the memory consumption rate to exceed the consumption constraint, then another query is chosen at random to decrease its allocation until the consumption rate is less than the memory size. Because of the shape of the curve of Figure 2.3, this may require more than one cycle of adjustment. Of course the result is still a candidate memory allocation and can be accepted or rejected by the annealing algorithm.

The size of the optimization problem can be reduced by simply having one variable per query which is the memory allocation. For a given memory allocation, each query may decide locally which join method to adopt for its minimum response time. The global effect of the trade-off of CPU processing and I/O is then neglected.

4. Heuristic Threshold Strategy

Next we try to observe simple rules of thumb for the way the global optimization strategy allocates memory and if a simple heuristic strategy therefore can be developed. To keep the situation manageable, we'll look at an example with hash join as the method of choice and examine how memory allocation to each query type changes under the optimization procedure as the query arrival rate changes.

Figure 4.1 Memory allocations

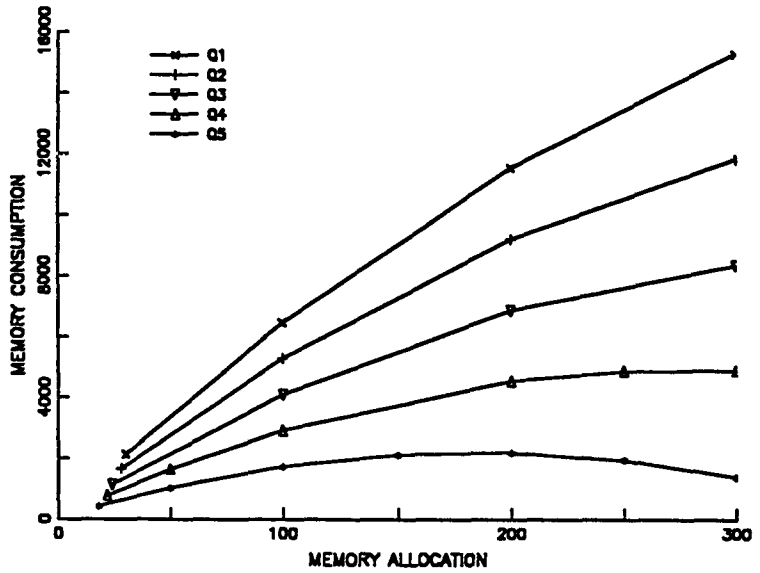


Table 4.1 Memory allocations

arrival rate (queries per sec)	0.035	0.045	0.055	0.065	0.067
Q1	1.000	0.170	0.100	0.100	0.100
Q2	1.000	1.000	0.260	0.090	0.090
Q3	1.000	1.000	1.000	0.150	0.080
Q4	1.000	1.000	1.000	1.000	0.470
Q5	1.000	1.000	1.000	1.000	1.000
Q6	1.000	1.000	1.000	1.000	1.000

Example 4.1: We now consider memory allocations among the following six queries which do hash joins on two relations:

Q1 joining two relations each with 60000 tuples over 600 pages
Q2 joining two relations each with 50000 tuples over 500 pages
Q3 joining two relations each with 40000 tuples over 400 pages
Q4 joining two relations each with 30000 tuples over 300 pages
Q5 joining two relations each with 20000 tuples over 200 pages
Q6 joining two relations each with 20000 tuples over 200 pages

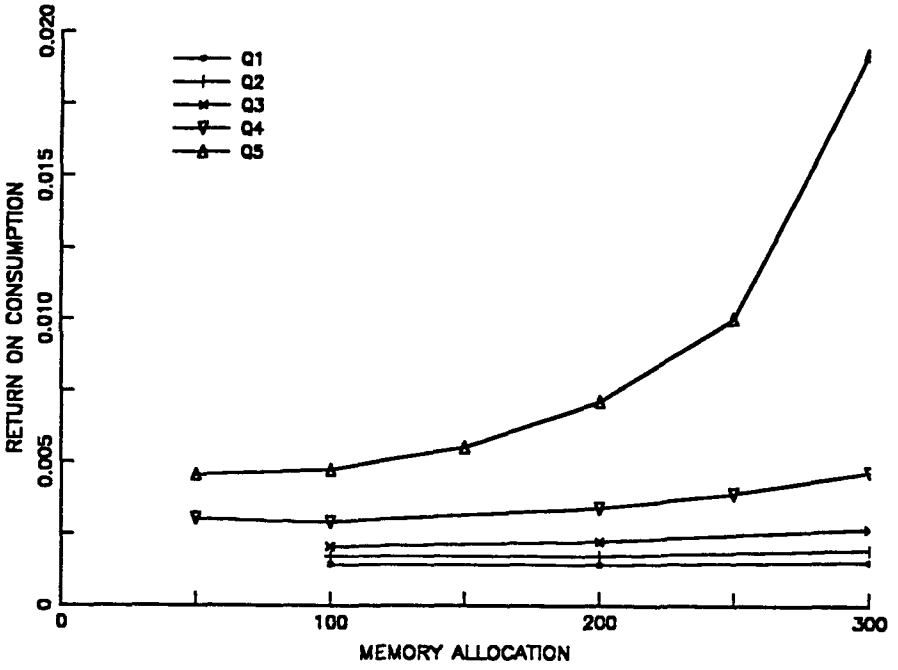
A join selectivity of 0.0001 is assumed for all queries. We further assume a memory buffer size of 300 pages with a system configuration specified in Table 2.1. Figure 4.1 shows the memory consumption versus memory allocation for the various types of queries and Figure 4.2 presents the ROC versus memory allocation. Notice Q6 has the same behavior as Q5, and hence is not explicitly shown. Query Q1 shows the largest memory consumption for a given allocation and the lowest ROC, whereas queries Q5 and Q6 are the opposite.

In Table 4.1, we show the memory allocations based on the global optimization procedure of Section 3 under different arrival rates for an arbitrarily chosen β of 0.75, α of 1, and δ of 1.25. The allocation is expressed relative to the maximum allowed allocation which is assumed to be 300 pages for all queries. At low query frequency, each query type is given its maximum allowed allocation. As the arrival rate increases, memory contention increases. Q1 is the first to be forced to run at its minimum allocation (31 pages). Further increase in the arrival rate forces Q2 and then Q3 to run at their minimum allocations (i.e., 28 and 25 pages, respectively). Q5 and Q6 with the maximum ROC continue to get their maximum allowed allocation at the expense of the other queries with lower ROC's. The large memory allocations are made to the queries with the smallest amount of tuples to join and the least response time. □

From the above example, we can make the following observations: When joining two relations of equal size under hash join, the processing and I/O saved by allocating an additional page of memory to a query is the same irrespective of the size of the relations to be joined if the relations have the same number of tuples per page (see Section 2). Therefore, ROC is maximized by allocating the memory page to the query with the shortest response time, because its memory consumption is the least. Looked at another way, for a given amount of memory consumption to be distributed among queries, more pages of memory can be allocated to a query with a larger ROC to improve response time. Thus, a global optimization strategy will not equally divide the memory among queries but will favor the queries with larger ROC for more memory allocation.

The optimal strategy divides the queries into three categories. The first category gets the maximum allowed allocation. (If the maximum requirement is less than the maximum allowed allocation per query, it will get the maximum required allocation.) The second category gets the minimum required allocation, which is the allocation with minimum consumption. The third category gets something in between. The majority of the queries fall into the first two categories. The third category is only for the borderline case. Furthermore, we can order the query types according to

Figure 4.2 Return on consumption



their ROC at the maximum allocation in descending order. Surprisingly, the net effect of the optimization strategy is to pick a dividing point on this ordered list. Elements preceding the dividing point fall into the first category, and elements following the dividing point fall into the second category. Elements at the dividing point, if any, belong to the third category.

Based on these observations we propose a simple heuristic strategy, referred to as the threshold strategy. It is based on ROC to determine how to allocate memory among concurrently executing queries. Assuming in-memory time of a join query is dominated mainly by I/O time, we can use the stand-alone execution time of a query as an estimate of its in-memory time.

Let us first consider the case where the join method for each query is predetermined by the query optimizer. Only the memory allocation needs to be determined in this case. Define for each query its *maximum allocation,* to be either its maximum required allocation or some predefined maximum allocation limit for all queries, whichever is smaller. For each query Q_i, let γ_i^{max} be its maximum ROC value within the maximum allocation. Recall that in Section 2, given the sizes of the joining relations and the join selectivity, the curve for ROC vs memory allocation can be directly derived. (We expect the query optimizer to provide the ROC information for each query step.) Thus, obtaining γ_i^{max} is straightforward.

The heuristic buffer management scheme attempts to allocate memory to maintain a certain level of ROC. It is important to look not only at the response-time reduction through additional memory allocation, but also at the associated cost (the amount of memory consumption), because the memory allocation does not reflect the effect of holding time. Define θ to be the threshold on ROC to be determined at the run-time environment. The single threshold parameter is applied to all query types to balance the global buffer allocation requirement. Under the threshold strategy, before a query is initiated, the amount of memory to be allocated for its execution is determined based on γ_i^{max} and θ. There is no attempt to change the allocation dynamically during query execution. (Only θ may be adjusted dynamically to affect the memory allocation of future queries.) For each query, say Q_i, if γ_i^{max} exceeds θ, the memory allocation corresponding to a ROC of γ_i^{max} will be provided before the query is initiated. Otherwise, the allocation with the minimum memory consumption is provided.

Next we consider the case in which both the join method and memory allocation are to be determined together. This can correspond to the case in which query access plans based on different join methods are pre-generated for each query. At execution time, depending on the value of θ, the suitable access plan is selected and the appropriate memory allocation is provided before the query is initiated. (Upon different activations of a given query, unless the query mix has changed drastically, most likely the optimal join method would not change, but the optimal memory allocation may change. The approach described below can be used to provide input to the query optimizer to generate an access plan of the desired join method and subsequently only the simplified technique described above would need to be used to decide the desirable memory allocation at run time.) As we observed in Section 2, different join methods have different memory consumption and minimum allocation requirements. For each query type, we can pick the the minimum memory consumption allocation over all join methods to calculate the ROC for each method. Note that changing the base of calculating the ROC will preserve the strict order on ROC values between any two allocations under a given join method. For example, under hash join, if originally the ROC at an allocation x is larger than that at an allocation y, using the minimum allocation of sort-merge join as the basis to calculate ROC will preserve this relation. This is due to the fact that if $a/b > c/d$ with $a > c$ and $b > f$, then $(a+\delta)/(b+\omega) > (c+\delta)/(d+\omega)$, for $\delta > 0$ and $\omega > 0$.

Thus for each query Q_i, let γ_{ij}^{max} be its maximum ROC value within the maximum allocation under the j-th join method. Let γ_i^{max} be the maximum of the γ_{ij}^{max}, for $j = 1, ..., 3$. Again a single threshold is applied to all query types to

balance the global storage allocation requirement. Under the threshold strategy, for each query, say Q_i, if γ_i^{max} exceeds θ, the corresponding join method is adopted and the associated memory allocation will be provided before the query is initiated. Otherwise, the join method and memory allocation with the minimum memory consumption are adopted.

5. Performance Comparison

In this section, we examine the performance of the heuristic strategy in Section 4 and the global optimization strategy in Section 3. Also considered is the fixed allocation strategy, which allocates each query with some predefined maximum allocation (or its maximum required allocation, if smaller). A simulation program is developed to compare the performance of the three strategies. The simulator consists of two parts. The first part simulates the different memory management strategies. Based on the strategy chosen, it decides the amount of memory allocation to the incoming query. For the threshold strategy, based on the threshold value and the query type which determines the ROC curve, the amount of memory allocation is determined. The second part is the system simulator which models the CPU and I/O queues and tracks the actual memory allocation. It keeps track of the progress of each activated query. Queries entering the system are activated if they can be given the entire memory allocation specified by the memory management strategy from an available memory pool (free page list); otherwise they remain on a wait list until their memory allocation is available. Queries are served on a FCFS basis. Both the CPU queue and I/O queues are FCFS. The service (CPU and I/O) demand of each type of query is determined by the query type. Queries run on the processor for a time period equal to the calculated CPU requirement divided by the calculated number of I/O, at which time they are switched to the disk queue. The relations are assumed striped or interleaved on the disks so that the disk utilizations are the same for all disks. After completion of I/O, the job returns to the CPU queue. At finish, the memory allocation for the job is returned to the available memory pool. All simulation runs are obtained such that the 95% confidence interval of the response time measure is estimated to be within 5% of the mean.

Example 5.1: There are 10 joins referencing 20 relations. Tables 5.1 and 5.2 show the pertinent parameters of the 20 relations and 10 join queries, respectively. Join selectivities are assumed to be 0.01 for query $Q7$, 0.0001 query for $Q1, Q9$, and $Q10$, and 0.00001 for all other queries. For hash join, δ is again assumed to be 1.5. The system configuration in Table 2.1 is assumed with a buffer size of 1000 pages.

Table 5.1. Relations in Example 5.1

Relation	Size (pages)	Cardinality (no. of tuples)	
1	500	4500	
2	500	4500	sorted on join attribute
3	700	50000	
4	700	50000	sorted on join attribute
5	800	80000	
6	800	80000	
7	1400	100000	
8	1400	100000	
9	650	40000	
10	650	40000	
11	400	30000	
12	400	30000	
13	2	10	
14	400	10000	
15	400	7000	
16	400	7000	
17	300	6000	
18	300	6000	
19	200	3000	
20	200	3000	

Table 5.2 Queries in Example 5.1

Query	Relations referenced		Relative run frequency
1	1	2	.032
2	3	4	.032
3	5	6	.036
4	7	8	.032
5	9	10	.032
6	11	12	.036
7	13	14	.180
8	15	16	.050
9	17	18	.200
10	19	20	.370

Figure 5.1 Response times under different strategies

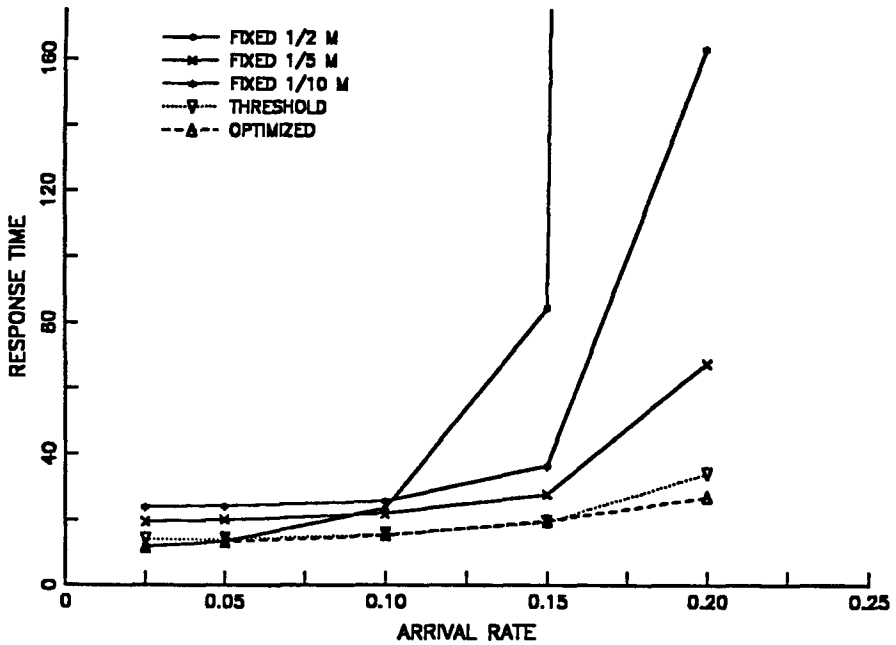
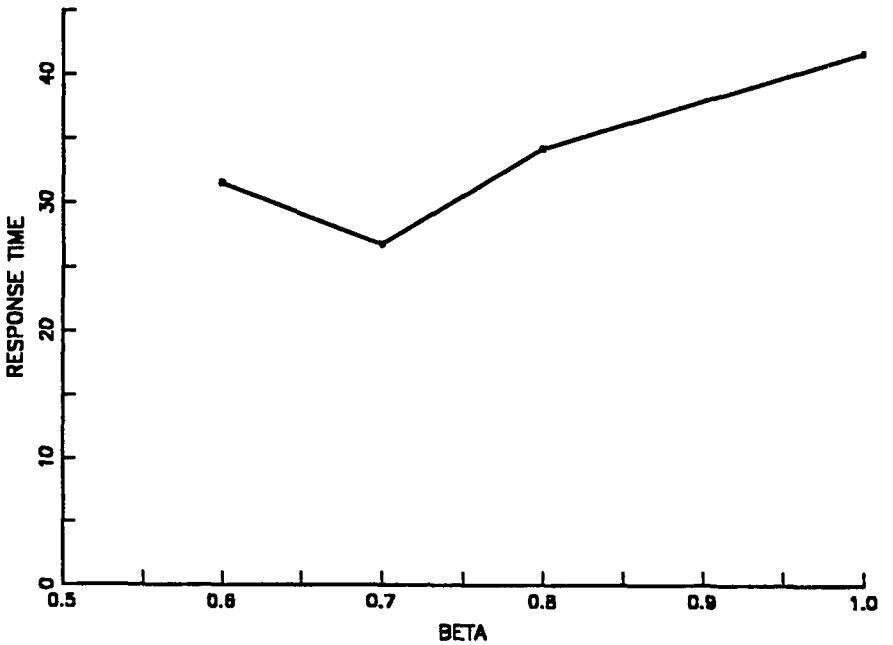


Figure 5.1 shows the response time versus query arrival rate for the optimized, the heuristic and the fixed allocation strategies. For the fixed allocation strategy, we consider three cases with the maximum allocation limits to be 1/2, 1/5, and 1/10 of the total buffer size, respectively. The heuristic threshold strategy performs quite close to the optimized strategy. The fixed allocation strategy does not do well over the whole range considered for all three cases. The fixed allocation with 1/2 of the buffer performs well at the low arrival rate but does badly as arrival rate increases. The fixed allocation with 1/10 of the buffer does badly at both low and high arrival rates. The fixed allocation with 1/5 of the buffer performs the best at the high arrival rate compared with the 1/2 and 1/10 allocations, but it is still much worse than the optimized and threshold strategies. As pointed out before, both these two strategies avoid allocating memory evenly, in favor of queries with higher ROC, thus leading to more robust performance. We have conducted many more simulations on different relative run frequencies, join selectivities, and relation sizes with similar results.

We now take a closer look at the global optimization strategy. First we examine its sensitivity to the parameters α, the maximum fraction of the buffer that can be allocated to a single query, and β, which sets the upper limit on total memory consumption of all queries. Figure 5.2 shows the average response time versus β

Figure 5.2 Optimal strategy's sensitivity to β



with an α of 0.5 and an arrival rate of 0.2 queries/sec. The optimal β is around 0.7. Too small a β causes memory to be wasted, i.e. not allocated, and too big a β results in long waiting time for the memory allocation to become available as memory usage becomes the bottleneck relative to the other resource like CPU. Next we examine the effect of α. Choosing α larger than 0.5 can seriously degrade the performance, because under the FCFS scheduling policy, the multiprogramming level (MPL) will be adversely affected. Consider the extreme case where $\alpha = 1$. For this case, understandably, when a query getting the maximum allocation is under execution, no other queries can get initiated. The execution time of that particular query is minimized but the CPU is idled most of the time with a MPL of one and the waiting time for the desired memory allocation increases substantially. Table 5.3 shows average response times for an optimized memory allocation for $\alpha = 0.5$ and $\alpha = 1$, respectively. For each α value, the optimal β is chosen. As expected, the response time for $\alpha = 1$ is extremely poor. We can see that the global optimization strategy does depend upon the appropriate selection of α and β. (It may be difficult to maintain "near optimal" settings for these parameters in a dynamically changing environment.) In Figure 5.1, the optimized strategy is plotted with $\alpha = 0.5$ and the optimal β derived through trial and error using

Table 5.3. Sensitivity to α

α	β	Response time
1.0	0.8	390.0
0.5	0.7	26.4

the simulation. The optimal value of β decreases as the arrival rate decreases to prevent memory from becoming the bottleneck.

The optimized strategies for the queries at an arrival rate of 0.2 queries/sec with $\alpha = 0.5$ and $\beta = 0.7$ are as follows:

Queries Q1 and Q2 do sort merge joins with minimum memory allocation.

Query Q7 does nested loops join with index scan on the inner relation R_{14}.

Queries Q3 and Q4 do hash join with minimum memory assignment.

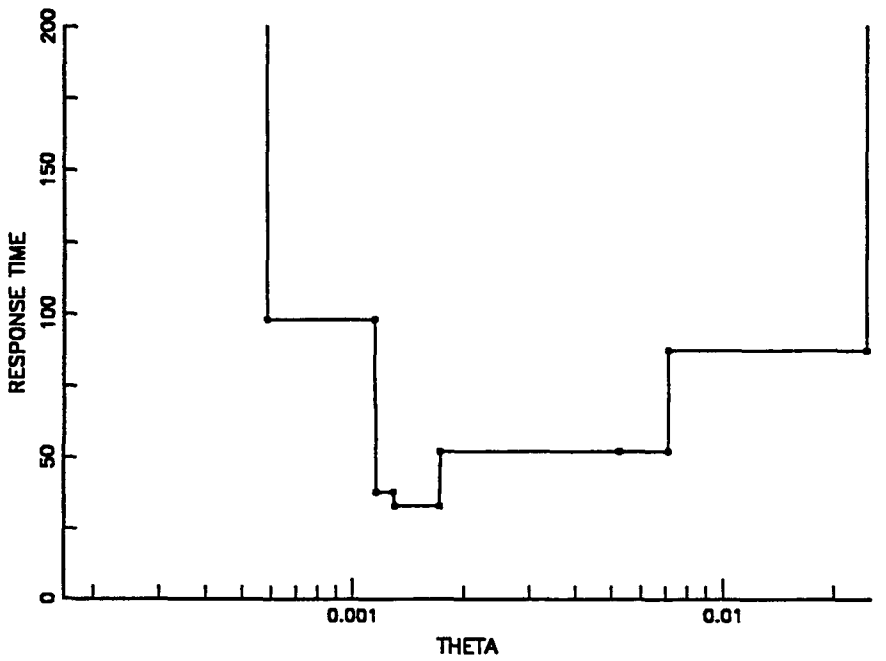
Query Q5 does hash join with 364 pages of buffer which is neither the maximum nor the minimum allocation.

Queries Q6, Q8, Q9, and Q10 do hash join with the maximum allocation.

The heuristic strategy with a maximum allocation constraint of half the buffer has a threshold of 0.0015, where the allocation is similar to the above except that Q5 is given the minimum allocation. Figure 5.3 shows the average response time versus the threshold, θ, for an arrival rate of 0.2 queries/sec. A smaller than optimal θ value means memory is over-allocated causing a longer wait time for the memory to be freed up, and a larger than optimal θ value means that the memory is under-allocated causing more memory resource to be wasted. Performance can be improved by properly setting θ at run time. In a quasi-stable environment, a dynamic approach can be taken to incrementally adjust θ and search for the optimal value. For example, if the memory wait time is too long, we can increase the value of θ by some increment. The average query's in-memory time will increase as more queries operate on their minimum allocations. However, the wait time for the required memory allocation to become available will decrease. If the net effect measured after some period of time does not deteriorate the query response time, which is the sum of the wait time and in-memory time, then θ is moving in the right direction and the procedure continues. Otherwise, θ will be adjusted in the opposite direction with a smaller decrement. The procedure stops if no performance improvement can be made by adjusting θ in either direction.

Simulations have been conducted for the dynamic approach of adjusting the θ value in Example 5.1. Both cases of starting θ with initial values larger than and

Figure 5.3 Threshold strategy's sensitivity to θ



smaller than the optimal range of values are considered. In either case, θ can move to the optimal range of θ values and similar average response times to that under the optimal θ can be achieved. □

Example 5.2 Tables 5.4 and 5.5 show the 20 relations and 10 join queries, respectively. Join selectivities are assumed to be 0.0001 for the first three queries and 0.00001 for the other queries. In this example, no relation is pre-sorted on the join column, hence hash join is the method of choice for all queries. The system configuration in Table 2.1 is again assumed with a buffer size of 2100 pages. Figure 5.4 shows the average response time versus query arrival rate for the heuristic threshold strategy and the fixed allocation strategy. For the fixed allocation strategy, we again consider the three cases with the maximum allocation limits to be 1/2, 1/5, and 1/10, respectively. The fixed allocation strategy does not perform well at higher arrival rate under all three cases. The threshold strategy with a maximum allocation constraint of half the buffer size operates under a θ of 0.00048 over the entire range. The optimized strategy is hardly distinguishable from the threshold strategy in this case, hence is not explicitly shown on Figure 5.4. Under the threshold strategy, all queries except queries Q9 and Q10 will get their maximum allocations whereas queries Q9 and Q10 only get their minimum allocations. □

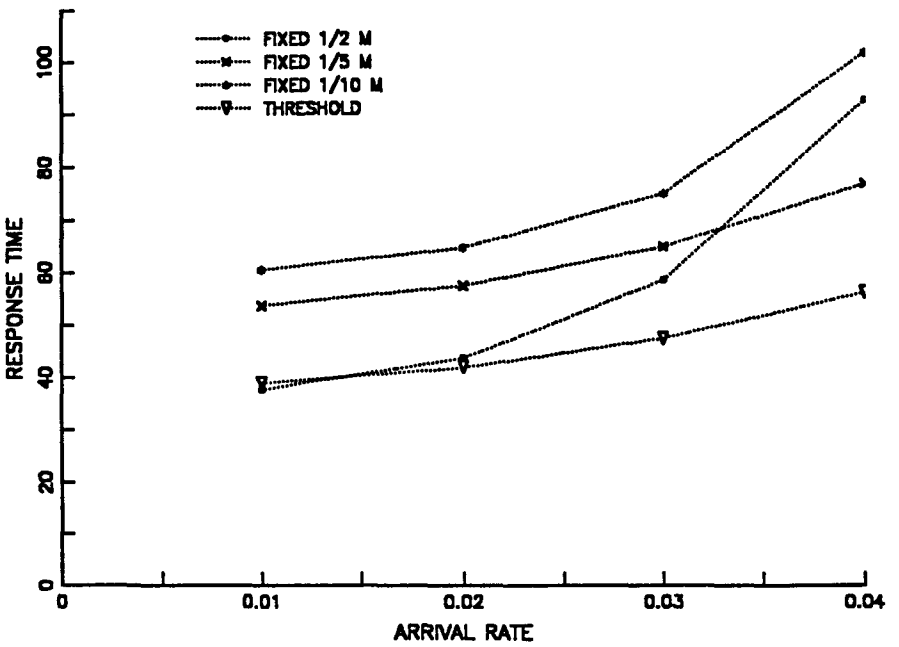
Table 5.4 Relations in Example 5.2

Relation	Size	Cardinality
1	200	10000
2	200	10000
3	300	15000
4	300	15000
5	500	30000
6	500	30000
7	700	50000
8	700	50000
9	800	75000
10	800	75000
11	1000	110000
12	1000	110000
13	1200	130000
14	1200	130000
15	1500	140000
16	1500	140000
17	1700	145000
18	1700	145000
19	2000	150000
20	2000	150000

Table 5.5 Joins in Example 5.2

Query	Relations referenced		Relative run frequency
1	1	2	.225
2	3	4	.215
3	5	6	.095
4	7	8	.095
5	9	10	.075
6	11	12	.075
7	13	14	.065
8	15	16	.065
9	17	18	.045
10	19	20	.045

Figure 5.4 Response times under different strategies



6. Conclusion

In a multi-query environment, the overall performance is very sensitive to buffer allocation strategies. This is especially the case when the relation size is larger than the buffer size. Conventional buffer management strategies do not provide guidelines on how much memory to allocate to the hash buckets under hash join or to the sort buffer like the tournament tree for sort-merge join in a multi-query environment. Allocating memory to optimize the performance of each query without considering the effect of other concurrently executing queries can lead to performance far from optimal. In this article, the concept of memory consumption and ROC is introduced as the basis for buffer management in a multi-query environment. Note that it is insufficient just to examine the reduction in I/O or response time as the criterion for allocating memory among contending queries. Even if two queries can result in the same I/O or response time reduction for the same amount of additional buffer allocation, the effect on memory consumption can be quite different as the two queries can have very different response times. Since the total buffer consumption must be preserved, it is important to introduce the concept of ROC. A global

optimization strategy is developed based on simulated annealing to provide a basis against which a more realizable algorithm can be devised and compared. By studying the optimal allocation, we observe that (1) the optimal strategy is not to evenly allocate the memory among queries like the fixed allocation strategy, and (2) the optimal strategy attempts to bias toward queries with larger ROC. Only those queries with larger ROC are given the maximum allocations, or allocation at the local maximum of the ROC curve, whereas most of the rest are given the minimum allocations. Furthermore, we expect the query optimizer should be able to estimate the response time curve (vs memory allocation), hence the ROC curve for a query. From these observations, a heuristic threshold strategy is proposed based on the concept of ROC. Simulation studies show that the heuristic strategy performs quite closely to the optimum and outperforms the fixed allocation strategy substantially.

Appendix

In this Appendix, we provide a simple analysis to estimate the amount of I/O and CPU processing under each join method.

A.1 Hash Join

Recall that the pathlength to perform an I/O operation is I_{io}, the pathlength to either extract each tuple from the input buffer, or move it to the output buffer is I_{move}, the pathlength to apply a hash function to the join attribute value is I_{hash}, the pathlength to search for a match or empty slot in the hash table is I_{search}, and the pathlength to join a pair of tuples is I_{join}. I_{search} may involve several comparisons or hash collisions. The number of comparisons is affected by the expansion factor, δ. For a fixed expansion factor, δ, the memory allocations will not affect I_{search} and will change only the portion of data to be read in twice. Define f to be the fraction of tuples in the first partition which is kept in the memory during phase one. Furthermore, let ψ be the join selectivity, i.e., the number of tuples resulting from the join divided by the product of the numbers of tuples in the two joining relations. Let $\{R\}$ and $\{S\}$ be the number of tuples in relation R and relation S, respectively.

The number of I/Os and the amount of CPU processing required are given below.

Phase 1 Cost

Number of I/Os

$$D^{phase\_1} = |R| + (1 - f)|R|.$$

The CPU processing is

$$U^{phase\_1} = |R|I_{io} + \{R\}(I_{move} + I_{hash}) + f\{R\}I_{search}$$

$$+(1 - f)(\{R\}I_{move} + |R|I_{io}).$$

Phase 2 Cost

Number of I/Os

$$D^{phase\_2} = |S| + (1 - f)|S|.$$

The CPU processing is

$$U^{phase\_2} = |S|I_{io} + \{S\}(I_{move} + I_{hash}) + f\{S\}I_{search}$$

$$+(1 - f)(\{S\}I_{move} + |S|I_{io}) + \{R\}\{S\}f\psi I_{join}.$$

Here we assume that the number of tuples matching from the join operation are spread over all the hash partitions and are proportional to the number of tuples in each partition.

Phase 3 Cost

Number of I/Os

$$D^{phase\_3} = (|R| + S)(1 - f).$$

The CPU processing is

$$U^{phase\_3} = (|R| + |S|)(1 - f)I_{io} + (\{R\} + \{S\})(1 - f)$$

$$(I_{move} + I_{hash} + I_{search}) + \{R\}\{S\}(1 - f)\psi I_{join}.$$

The total number of I/O accesses is the sum of the I/O accesses in the three phases. Similarly, the total number of instructions processed by the CPU is the sum of the CPU processing in the three phases. Note that we have been ignoring the effect of prefetch blocking, but the extension is straightforward.

A.2 Sort-Merge Join

Under sort-merge join, the CPU requirement is composed of the following operations.

1. Set up the I/O operation
2. Move tuples from the I/O buffer to the tournament tree
3. Sort/Merge tuples through the tournament tree
4. Move tuples from the tournament tree to the output buffer
5. Compare the join fields of tuples in the two sorted relations to perform the join.

Both the scan phase and merge phase go through the first four steps and the final join phase involve steps 1 and 5. The CPU requirement at each step except Step 3 is some constant, which is independent of the memory allocations, $\times$ the number of tuples in each relation. Step 3 depends upon the size of the tournament tree.

Assume the pathlengths to move the tuple with the next smallest join attribute value out of the tournament tree and refill the tree is I^s_{tour} and I^m_{tour} for the scan phase and merge phase, respectively. I^s_{tour} or I^m_{tour} is roughly equal to the number of levels in the tournament tree multiplied by I_{comp}, the pathlength to perform a comparison operation at each level. The number of levels in the scan phase is determined by the memory allocation and that in the merge phase is determined by the number of sorted runs generated in the scan phase. The number of I/Os and the amount of CPU processing required are derived below.

Phase 1 Cost

Number of I/Os

$$D^{phase\_1} = 2(|R| + |S|).$$

The CPU processing is

$$U^{phase\_1} = 2(|R| + |S|)I_{io} + (\{R\} + \{S\})(2I_{move} + I^s tour).$$

Phase 2 Cost

Number of I/Os

$$D^{phase\_2} = 2(|R| + |S|).$$

The CPU processing is

$$U^{phase\_2} = 2(|R| + |S|)I_{io} + (\{R\} + \{S\})(2I_{move} + I^m_{tour}).$$

Phase 3 Cost

Number of I/Os

$$D^{phase\_3} = |R| + |S|.$$

The CPU processing is

$$U^{phase\_3} = (|R| + |S|)I_{io} + (\{R\} + \{S\})(I_{move} + I_{comp}) + \{R\}\{S\}\psi I_{join}.$$

A.3 Nested-Loop Join

Assume the I_{proc} to be the pathlength to scan and compare a tuple. I_{proc} is the sum of I_{move} and I_{comp}. Assume that R is the inner relation. Define g to be $M/|R|$, i.e., the portion of the inner relation that can be kept in memory. If the entire inner relation can be kept in memory, the number of I/O's is shown by

$$D = |R| + |S|.$$

The CPU processing overhead is,

$$U = (|R| + |S|)I_{io} + \{R\}\{S\}I_{proc} + \{R\}\{S\}\psi I_{join}.$$

Next we consider the case where neither relation can be kept entirely in memory.

For table scan under LRU,

$$D = |S||R| + |S|$$

$$U = (|S||R| + |S|)I_{io} + \{R\}\{S\}I_{proc} + \{R\}\{S\}\psi I_{join}.$$

For table scan under MRU,

$$D = (1 - g)|S||R| + |S|$$

$$U = ((1 - g)|S||R| + |S|)I_{io} + \{R\}\{S\}I_{proc} + \{R\}\{S\}\psi I_{join}.$$

For index scan, assuming no buffer hit at the data pages, (see Mackert and Lohman, 1989 for more elaborate estimations),

$$D = |S| + \psi\{S\}\{R\}$$

$$U = (|S| + \psi\{S\}\{R\})I_{io} + \{R\}\{S\}\psi(I_{proc} + I_{join}).$$

Acknowledgments

The authors would like to thank Joel W. Wolf for his comments and suggestions.

References

Chou, H.-T. and DeWitt, D.J., An Evaluation of Buffer Management Strategies for Relational Database Systems, *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, 1985.

Codd, E.F. Relational Model of Data for Large Shared Data Banks, *Communications of the ACM* 13:377-387, 1970.

Cornell, D.W. and Yu, P.S. Integration of Buffer Management and Query Optimization in Relational Database Environment, *Proceedings of the 15th International Conference on Very Large Data Bases,* Amsterdam, Netherlands, 1989.

Dewitt, D.J., Katz, R.H., Olken, F., Shapiro, D.L., Stonebraker, M.R., and Wood, D. Implementation Techniques for Main Memory Database Systems, *Proceedings of the SIGMOD International Conference on Management of Data,* Boston, MA, 1984.

Effelberg, W. and Loomis, M.E.S. Logical, Internal and Physical Reference Behavior in CODASYL Database Systems, *ACM Transactions on Database Systems,* 9:187-213, 1984.

Ioannidis, Y. and Wong, E. Query Optimization by Simulated Annealing, *Proceedings of ACM SIGMOD International Conference on Management of Data,* San Francisco, CA, 1987.

Ioannidis, Y. and Kang, Y. Randomized Algorithms for Optimizing Large Join Queries, *Proceedings of ACM SIGMOD International Conference on Management of Data,* Atlantic City, NJ, 1990.

Ioannidis, Y. and Kang, Y. Left-Deep vs Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization, *Proceedings of ACM SIGMOD International Conference on Management of Data,* Denver, CO., 1991.

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. Optimization by Simulated Annealing, *Science,* 220:671-680, 1983.

Kirkpatrick, S. and Tolouse, G. Configuration Space Analysis of Travelling Salesman Problems, *Journal Physique,* 46:1277-1292, 1985.

Knuth, D.E., *The Art of Computer Programming: Vol. 3,* Reading, MA: Addison-Wesley, 1975.

Lakshmi, M.S. and Yu, P.S. Limiting Factors of Join Performance on Parallel Processors, *Proceedings of the Fifth International Conference on Data Engineering,* Los Angeles, CA, 1989.

Mackert, L.F. and Lohman, G.M. Index Scans Using a Finite LRU Buffer: A Validated I/O Model, *ACM Transactions on Database Systems,* 14:401-424, 1989.

Ng, R., Faloutsos, C., and Sellis, T. Flexible Buffer Allocation Based on Marginal Gains, *Proceedings of ACM SIGMOD International Conference on Management of Data,* Denver, CO, 1991.

Sacco, G. and Schkolnick, M. A Mechanism for Managing the Buffer Pool in a Relational Database System using the Hotset Model, *Proceedings of the Eighth International Conference on Very Large Data Bases,* Mexico City, 1982.

Sacco, G. and Schkolnick, M. Buffer Management in Relational Database Systems, *ACM Transactions on Database Systems,* 11:474-498, 1986.

Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., and Price, T.G. Access Path Selection in a Relational Database Management System, *Proceedings of ACM SIGMOD International Conference on Management of Data,* Boston, MA, 1979.

Shapiro, D.S. Join Processing in Database Systems with Large Memories, *ACM Transactions on Database Systems,* 11:239-264, 1986.

Stonebraker, M. Operating System Support for Database Management, *Communications of the ACM,* 24:412-418, 1981.

Swami, A. and Gupta, A. Optimization of Large Join Queries, *Proceedings of ACM SIGMOD International Conference on Management of Data,* Chicago, 1988.

Swami, A. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques, *Proceedings of ACM SIGMOD International Conference on Management of Data,* Portland, OR, 1989.

Wolf, J.L., Dias, D.M., Iyer, B.R., and Yu, P.S. Multisystem Coupling by a Combination of Data Sharing and Data Partitioning, *IEEE Transactions on Software Engineering,* 15:854-860, 1989.