Dynamic Resource Brokering for Multi-User Query Execution

Diane L. Davison†
Informix Software, Inc.
davison @ informix.com

Goetz Graefe‡
Microsoft Corp.
goetzg @ microsoft.com

Abstract

We propose a new framework for resource allocation based on concepts from microeconomics. Specifically, we address the difficult problem of managing resources in a multiple-query environment composed of queries with widely varying resource requirements. The central element of the framework is a resource broker that realizes a profit by "selling" resources to competing operators using a performance-based "currency." The guiding principle for brokering resources is profit maximization. In other words, since the currency is derived from the performance objective, the broker can achieve the best performance by making the scheduling and resource allocation decisions that maximize profit. Moreover, the broker employs dynamic techniques and adapts by changing previous allocation decisions while queries are executing. In a first validation study of the framework, we developed a prototype broker that manages memory and disk bandwidth for a multi-user query workload. The performance objective for the prototype broker is to minimize slowdown with the constraint of fairness. Slowdown measures how much higher the response time is in a multi-user environment than a single-user environment, and fairness measures how even is the degradation in response time among all queries as the system load increases. Our simulation results show the viability of the broker framework and the effectiveness of our query admission and resource allocation policies for multi-user workloads.

1. Introduction

In order to maximize system performance in a multi-user query environment, it is necessary that resources such as memory, disk bandwidth, and processor bandwidth be allocated effectively during query execution. Since resources are inter-related, it is desirable for the resource manager to handle all resources that affect performance. For example, memory allocated to a memory-intensive operator such as a hash join allows the operator to use a larger hash table, and therefore reduces disk contention. A resource manager that takes both memory and disk bandwidth resources into account can balance memory and disk contention and, as we will show, provide better performance. At any point in time, a resource manager makes the best possible allocation decisions based on the current workload, however, resource contention may change as queries enter and leave the system. To achieve the best performance, a resource manager should be capable of adapting to these changes in resource contention. Adaptability could be achieved by changing future allocation decisions based on the current resource contention [MeD93a, Yu93], or as we will show, by dynamically changing current allocation decisions, necessitating operators that can adapt to fluctuations in their resource allocation while they are executing.

We propose a new framework for query scheduling and resource allocation that uses concepts from microeconomics to reduce complexity. The framework presently addresses shared-memory architectures, but we plan to extend it to hierarchical architectures [GrD93]. After a performance objective has been defined, a currency is derived that reflects the value of a resource allocation to the objective. A resource broker, the central element of the framework, sells resources to query processing operators in the currency in order to achieve the performance objective. The broker realizes a profit by selling resources, and maximizing profit results in the best performance because of the relationship of the currency to the performance objective. Thus, the principle of *profit maximization* guides the broker in all of its admission and allocation decisions. The operators submit bids to the broker to buy the system resources they need to execute, such as memory, disk bandwidth, and processor bandwidth. The key elements of the broker are the *currency*, which is used by the allocation policy to determine the value of an allocation toward the performance objective; the *admission policy*, which controls the number of current bidders; and the *allocation policy*, which controls how the scarce resources are shared among the current bidders.

The *currency* is derived from the performance objective so that the bid price is a direct reflection of the value of an allocation, where a higher price indicates an allocation that is more favorable to the objective. Since a higher bid price necessarily implies a larger overall contribution to the performance objective, the broker can attain the best performance simply by making the scheduling and resource allocation decisions that maximize its profitability. While previous microeconomic techniques for resource allocation use an artificial currency for the sale of resources and do not address the distribution of wealth among bidders or users [FYN88, WHH92], our performance-based currency effectively provides an implicit distribution of wealth that maximizes system-wide performance.

Scheduling new queries is controlled by the *admission policy*. A new query will be admitted only if doing so will increase broker profitability. Clearly, a new query may be admitted if sufficient resources are available to meet its needs. A new query may also be admitted if sufficient resources could be bought back from lower bidders to meet the needs of the new query; this increases profit and thus performance. Once a query gains admission, the operators in the query are allowed to bid for resources according

This research was partially supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland (UMIACS), Digital Equipment Corp., and Texas Instruments.

† Diane L. Davison performed this work at the University of Colorado at Boulder.

‡ Goetz Graefe is currently on leave from Portland State University.

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

to the allocation policy.

The *allocation policy* determines how the scarce resources are shared among the currently executing operators, with resources being sold to the highest bidder, or operator, in order to maximize profit and therefore overall system performance. Executing operators are required to submit bids to the broker multiple times during their execution, where the result of a bid is the amount of resources the bidder is allowed to use. The allocation resulting from a bid may be an increase, decrease, or no change from the bidder's previous allocation, and the operator is expected to adapt to any changes in its allocation This is accomplished by using adaptable algorithms, such as [DaG94, PCL93a]. Bidding during operator execution allows the broker to dynamically adapt to changes in resource contention In other words, the broker may buy back resources from a low bidder to sell to a higher bidder to increase profitability. The broker avoids delays by quickly responding to each bid with an allocation of resources. Previous allocation techniques have used auctions to sell resources [FYN88, WHH92]. In an auction, the broker collects bids until a certain amount of time has passed; the winner of the auction is then chosen from among the bidders. Auctions perform well for coarse granularity resource allocation, but the delay due to bid collection may be too large for fine granulity resource allocation.

The remainder of this paper is organized as follows. In Section 2, we briefly review previous work that is related to our research. Section 3 presents the framework for resource brokering, while the specific policy decisions we made for our prototype broker are detailed in Section 4. We describe the simulator we used for experimentation in Section 5, and present our experimental results in Section 6. Finally, we summarize the paper and offer our conclusions.

2. Related Work

There are three areas of previous work that are relevant to the research presented in this paper. microeconomics techniques for resource allocation, memory allocation techniques, and adaptable algorithms.

Ferguson et al. proposed using microeconomic agents to manage computer resources in large, distributed systems [FYN88]. A load balancing economy was studied in which jobs entering the system were given money to bid for processor time and communication capacity through auctions. By buying communication capacity, a job could migrate among nodes in order to obtain a lower price for processor time The economy was shown to improve performance compared to a non-economic load balancing algorithm. This research demonstrated that competitive microeconomic techniques are generally useful for resource allocation problems in computer systems Furthermore, the microeconomic approach reduced the complexity of resource allocation problems and provided stability. The paper did not propose specific policies to control the distribution of wealth, but the goal was that wealth would be used as a priority so that wealthy jobs would experience lower response times than poor jobs. The use of wealth as a priority was shown to be ineffective when the bidder changes its bid price due to losing past auctions. The two disadvantages of the auction process are that it has a potentially high overhead due to the bid collection process, and that the sale price of a resource does not accurately reflect the demand for the resource because the highest bidder pays its bid price regardless of the amount of resource contention, or number of bidders. We address the distribution of wealth with a performance-based currency, which provides an implicit distribution of wealth that maximizes system-wide performance. The broker avoids the delays inherent to an auction by responding quickly to each bid with an allocation of resources.

Waldspurger et al. created an economy to utilize idle CPU time in a distributed network of heterogeneous workstations [WHH92]. SPAWN uses auctions to sell idle CPU time on workstations, and the winner is granted exclusive use of the machine for the duration of the time slice. The time slice is typically of a coarse granularity, e.g., one minute. To perform work, a task may spawn child tasks, which are funded by the parent. Additional child tasks may be spawned given sufficient funding and resource availability, and child tasks that are no longer cost effective may be terminated. SPAWN uses funding rates, rather than absolute distributions of wealth, and the allocation of funding to users is assumed to be handled by human administrators.

Stonebraker et al. propose to use a microeconomic model for storage management and query execution in the Mariposa distributed database system [SDK94]. Each query has a budget with which to bid for services. Queries are divided into subqueries, each of which receives a portion of the budget. A "broker" is responsible for getting the query performed by finding the appropriate sites to execute the subqueries. The paper does not discuss the method to execute the subqueries once they are assigned to a site, which is the problem we address in this paper.

Return on Consumption (ROC) was proposed by Yu and Cornell to measure the effectiveness of additional memory allocation on overall response time improvement [Yu93]. ROC is the benefit to cost ratio of additional memory compared to an operator's minimum memory allocation For hash join, the minimum memory allocation is $M_{\min} = \sqrt{fudgeFactor \times R}$, where R is the size of the build input in pages, and *fudgeFactor* is used to account for hash table overhead. A heuristic strategy is proposed and shown to be near optimal. Essentially, the most effective use of memory to reduce average response time is to give the maximum memory allocation to small operators and the minimum memory allocation to large operators. However, the heuristic strategy is very sensitive to its target ROC parameter that determines the division between small and large operators, and the paper does not elaborate how to set this parameter. We use ROC as the currency for our prototype broker.

Carey et al. used *fairness* in a study of dynamic query allocation in distributed database systems [CLL85]. Fairness is achieved if the ratio of response time to wait time is the same for all queries. Mehta and DeWitt present scheduling and allocation policies for complex workloads to satisfy the goal of fairness [MeD93a]. Fairness is defined in [MeD93a] as an even degradation in response time among different classes of queries as the system load increases. An adaptive technique is proposed in which queries are categorized into three classes according to the size of the build input. Separate per-class multiprogramming levels (MPLs), which may be dynamically adjusted, control the query scheduling. Queries in the small class are allocated their maximum memory allocation, while queries in the large class are allocated their minimum memory allocation. The remaining memory is evenly allocated among the queries in the medium class. Fairness is measured as follows. First, determine for each class the ratio of the average observed response time to the average standalone response time (the response time when the query is executed alone in the system with all resources at its disposal). This ratio is the *slowdown*, which measures how much higher the response time is in a multi-user environment than in a single-user environment. Fairness is then defined in [MeD93a] as the standard deviation in the three ratios, so a lower value indicates more fair treatment. The adaptive technique is compared to several other static scheduling and allocation techniques and shown to provide much better fairness. This research identified the important criterion of "fairness" for a complex, multiple-query workload and demonstrated the ineffectiveness of static allocation and scheduling strategies (such as FCFS scheduling) to attain this

goal. This research used fairness as the goal, but a system could meet the goal of fairness while providing very high response times. Therefore, the performance objective for our prototype broker is to minimize slowdown with the *constraint* of fairness over all queries.

Pang et al. proposed dynamic memory allocation strategies for real-time database systems with firm deadlines [PCL94]. The objective of Priority Memory Management (PMM) is to minimize the number of missed deadlines, and it does this by keeping resource utilization high to fully exploit the machine, and by effectively allocating memory. PMM dynamically adjusts the multiprogramming level (MPL) to keep resource utilization high. In other words, more queries may be permitted to run with less memory if disk utilization is low. To effectively allocate memory, PMM incorporates the results from [Yu93]. One of two memory allocation techniques is used at any point in time: Max gives the maximum memory allocation to all executing queries, and MinMax gives the maximum allocation to urgent queries and the minimum allocation to the rest. As a query nears its deadline, it becomes urgent and its allocation will be fixed at its maximum memory requirement. Thus, the memory allocation made to a query may dynamically vary among no allocation, its minimum, and its maximum. Compared to static and proportional allocation schemes, PMM was shown to be much more effective in minimizing the number of missed deadlines, especially under heavier system loads. This work is the only work of which we are aware that dynamically varies operator memory allocations, as does the broker. Although the techniques presented could be adapted for more general use, the PMM algorithm is specific to real-time systems with firm deadlines. In our research, we propose a new and general framework for resource allocation that supports a wide variety of objectives. In our prototype broker, we are also concerned with improving device utilization, but we take a different approach than increasing the MPL. The broker allocates *both* memory and disk bandwidth to improve resource utilization.

We designed a new adaptable hash join algorithm, the Memory-Contention Responsive Hash Join (MCRHJ) [DaG94], and found in our validation study that MCRHJ provides lower response times than competitive algorithms, such as [PCL93b]. Therefore, the MCRHJ algorithm is used in our prototype broker. MCRHJ achieves a reduction in the amount of *time* spent on I/O and its adaptability by using dynamically sized I/O buffers, or *clusters*. A reduction in I/O time requires a tradeoff between the I/O volume (number of pages of intermediate I/O) and the number of I/O operations. The basic algorithm divides the build and probe inputs into multiple partitions, any of which may be resident in memory or spilled to disk. Pages not in use by any partition are kept on a free list. Resident partitions may obtain pages from the free list to increase the size of the hash table, and spilled partitions may obtain pages to enlarge their output clusters. When the free list is exhausted, either a spilled partition must be flushed or a resident partition must be spilled. The output cluster size is maximized by choosing the largest partition to flush or spill. An increase in memory is exploited by dynamically increasing the size of the I/O clusters, and by restoring spilled partitions to memory. A decrease in memory is adapted to by dynamically decreasing the size of the clusters, and by spilling resident partitions.

3. Resource Broker Framework

The broker framework provides the capability to dynamically manage the allocation of resources among resource-intensive queries in order to achieve a system-wide performance objective. To simply this complex problem, concepts from microeconomics are exploited. Furthermore, the framework supports a wide variety of objectives by modification of a small set of policies. The broker manages both the admission of queries to the system, which affects resource contention, and the allocation of resources among executing operators, which uses a bidding process in a performance-based currency. The currency is derived from the performance objective so that it directly reflects the performance impact of an allocation, allowing the broker to maximize performance automatically by simply maximizing profit. The interaction of queries and operators with the broker is illustrated in Figure 1. Once a new query is issued by a user, it requests admission, or the right to execute, from the resource broker. Depending on the admission policy and the system load, the query may be delayed before admission is granted, but a query is never denied admission. When the query does gain admission to the system, the operators in the query are permitted to bid for resources. The broker controls all resources and holds unallocated resources in short-term reserves (other possibilities for structuring the broker are discussed at the end of this section). If the reserves are insufficient to meet the bidder's resource requirements, the bidder must wait until the reserves have been replenished. To replenish the resource reserves, the broker may buy resources back from a bidder with a lower bid price than the waiting bidder. This is the same mechanism the broker uses to dynamically adapt to changes in resource contention. In other words, the broker automatically adapts to changes in the workload because of its price system. Note that the broker controls all allocation decisions, so bidders are required to sell or buy resources as directed by the broker; however, the broker will never require a bidder to reduce its allocation below its minimum memory requirement. This allows the broker to maximize overall system performance by selling the most resources to queries that can make the largest contribution to overall performance. Furthermore, the queries and operators are trusted by the broker. The five elements of the broker are: (1) performance objective, (2) performance-based currency, (3) admission policy, (4) allocation policy, and (5) adaptable algorithms. We now describe each of these five elements at a conceptual level. Although we describe the broker as if it were a single entity, it could be implemented as a hierarchy or some other structure, and we discuss such issues at

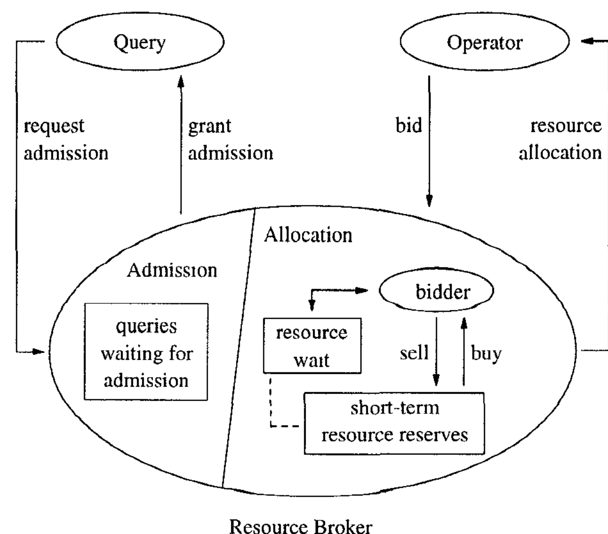

Figure 1. Broker Framework.

the end of this section.

The goal of the broker is to achieve a system-wide *performance objective*, where any objective may be specified, such as "minimize average response time," "maximize throughput," or some other goal. Once the performance objective has been determined, it is necessary to derive a currency that directly reflects the value to this objective of an allocation. This derivation of the currency from the performance objective means there is a direct association between broker profitability and system-wide performance. This simplifies the broker's task, allowing it to focus only on maximizing profit, and the system-wide performance objective is automatically maximized.

The *currency* is used by the allocation process to determine which bidder may buy what amount of resources. During execution, operators submit bids in the currency to buy resources from the broker, and the allocation policy dictates how resources should be divided among the competing bidders. The currency measures the contribution of the bidder to the system-wide performance objective, and a higher bid price necessarily implies a larger contribution. In fact, the currency and expected performance must be related linearly. When defining the currency, it may be necessary to take into account the *opportunity cost* [GIL89]. Considering the benefit of an allocation to an operator (e.g., relative reduction in response time) accounts for the local effect of the allocation. However, there is also a cost associated with every allocation in that the opportunity is sacrificed to allocate those resources to any other operator. Ignoring the opportunity cost may result in a Pyrrhic allocation.

The *admission policy* controls resource contention by scheduling new queries for execution. There are two issues for the admission policy. First, the next query to execute must be chosen from among the waiting queries. This could be accomplished using a simple FCFS queue, or a more sophisticated out-of-order technique. Second, the policy must dictate when the new query should be admitted. This may, for example, necessitate achieving a certain level of resource availability before the query may execute. The policy must be profit-driven, so that a new query will be admitted if doing so will increase broker profitability. For example, the broker may admit a new query whose operators could buy sufficient resources from currently executing operators with lower bid prices.

When a query is scheduled, its operators become eligible to bid for resources under the control of the *allocation policy*. Each bidder, or operator, is guaranteed some resource allocation; the amount of this allocation is a policy decision but is at least the minimum amount of resources that the operator requires to execute. If this amount of resources is not available, the bidder must wait until sufficient resources become available. Remaining resources are then sold to the highest bidder, accomplishing two important things. First, the broker always makes the best use of resources as they contribute to the performance objective. Second, resources are fully exploited since idle resources will be sold to lower bidders if they cannot be used by higher bidders (i.e., higher bidders have their maximum allocation). To maximize the utility of resources, the broker does not explicitly maintain a long-term reserve of resources; however, it may temporarily hold resources in reserve to adapt to changes in the workload that occur as new queries enter and old queries leave the system. That is, the broker may buy back resources from one bidder to sell to another; these resources are temporarily held in reserve until they are sold.

The broker dynamically adapts to changes in the workload by adjusting previous resource allocation decisions, necessitating *adaptable algorithms*. The operators must bid for resources multiple times during their execution, where the result of a bid may be an increase, decrease, or no change in the operator's previous allocation. The operators must be capable of gracefully adapting to these resource fluctuations by immediately adjusting their resource usage as directed by the broker. Note that the broker will never require an operator to reduce its allocation below its minimum requirement nor allow it to exceed its maximum requirement.

Broker Implementation Issues

We have described the broker conceptually, and we now briefly discuss some implementation issues. The broker could be implemented as a single entity, or a collection of entities, depending on considerations such as the machine architecture, workload, and the performance objective. For a shared-memory machine with a simple objective, a single broker could manage all resources (memory, disk bandwidth, and processor bandwidth). The broker could be implemented as a shared data structure or a separate process. For a more complex architecture or objective, a hierarchical structure may be used, where first-level brokers manage bidding among queries, and second-level brokers manage bidding among first-level brokers. Furthermore, a different currency may be used by each broker, depending on its objective. This may require the definition of an inter-currency exchange rate.

A two-level hierarchy of brokers might be appropriate to manage resources in a hierarchical architecture, which is a distributed-memory machine with shared-memory nodes [GrD93]. Each node could have a broker to manage intra-node resources (i.e., memory, disk bandwidth, and processor bandwidth), and a global broker could manage global resources, such as interconnection bandwidth, and perform load balancing. For a large machine, the global broker might be several brokers, each responsible for a subset of the nodes.

As an example of a structure for a complex workload, consider a multiclass query workload, where each class has a different performance objective [BMC94]. For this workload, one broker could manage the resources assigned to each class. A global broker could then manage the allocation of resources among the classes by selling resources to the per-class brokers depending on how near they are to their performance objectives. Note that [BMC94] specifically considers a mix of transaction and query classes, while the broker addresses query workloads but not transactions.

4. Prototype Resource Broker

In the previous section, we described a framework for dynamic resource brokering at the conceptual level. In a first validation study of this framework, we have implemented a prototype broker to manage memory and disk bandwidth for a multi-user workload composed of hash join queries with widely varying resource requirements. The broker explicitly controls the amount of disk bandwidth used by individual queries by allocating disk I/O buffers for asynchronous I/O. The prototype broker is implemented as a shared-memory data structure, rather than as a separate process or thread. In this section, we describe in detail the specific policies that we defined for this prototype broker, covering each of the five elements of the framework.

4.1. Performance Objective

The performance objective is to minimize slowdown with the constraint of fairness. Slowdown measures how much longer the multi-user response time is for a query than the single-user response time. Fairness is an even degradation in slowdown for all queries regardless of size as the system load increases. Our definitions of these metrics differ slightly from those in [MeD93a], so we describe them here.

For our calculation of the two metrics, we consider each query individually, rather than categorizing the queries into classes. This provides the most accurate slowdown and fairness values and handles high variations in slowdown among all queries. The average per-class standalone response times are used to normalize the average per-class observed response times in [MeD93a]. Since response times may vary significantly *within* a class, we normalize the individual response times before taking the mean. *Slowdown* is determined as follows. For each query, determine the ratio of the observed response time to the standalone response time for that query. Slowdown is the mean of the ratios for all the queries. A lower slowdown is desirable since it means that the response times were closer to the standalone response times. *Fairness* is the standard deviation in all of the individual slowdown values. A lower fairness value is preferable, since it means that the queries degraded more evenly in performance. It is important to note that our method of calculating fairness will result in higher fairness values than the class method, since the class method eliminates variations in response time by averaging.

It is important to point out that slowdown and fairness are only used to measure the effectiveness of our policies, and are not an integral part of the broker itself. The adaptive policy proposed in [MeD93a] has the goal of achieving the best fairness, and explicitly uses fairness to guide scheduling and allocation decisions. Therefore, it would not have been feasible for that policy to determine fairness according to our definition (since it is impossible to have available the individual standalone times for all possible queries in a real system). Our intent is to show that the brokering policies we have defined meet the goal of minimizing slowdown with the constraint of fairness; the slowdown and fairness values need not be calculated in an actual implementation.

4.2. Currency

The currencies for both memory and disk bandwidth allocation are based on Return on Consumption (ROC), which was devised by Yu and Cornell to measure the effectiveness of additional memory consumption on response time improvement [Yu93]. ROC is ideally suited as a broker currency because it satisfies the important criteria discussed earlier. ROC is directly derived from the objective of reducing response time; it provides a linear relationship between profitability and performance; and it incorporates the opportunity cost of an allocation. Specifically, ROC is the *benefit / cost* ratio of allocating additional memory to an operator, and is defined as:

$$ROC(M) = \frac{benefit}{cost} = \frac{T(M_{\min}) - T(M)}{M \times T(M) - M_{\min} \times T(M_{\min})},$$

where $M > M_{\min}$ is the operator's memory allocation, $M_{\min}$ is the operator's minimum required memory allocation, and $T(M)$ is the estimated response time of the operator with a memory allocation of M. The calculation of $T(M)$ assumes random I/O and accounts for large block I/O by assuming an average I/O block size of half the best block size for the system (the actual block size dynamically varies, as discussed in [DaG94]). Due to limited space, we do not include the analysis for estimation of $T(M)$, but it may be found in [Dav95]. In the *ROC* formula, the benefit is the decrease in response time resulting from the additional memory allocation compared to the response time with the operator's minimum memory requirement. The cost is the increase in memory consumption (the space-time product) resulting from the additional memory. Yu and Cornell determined that the most effective use of memory to reduce average response time is to give the maximum memory allocation to small operators and the minimum memory allocation to large operators. To determine *ROC*, it is necessary to reevaluate optimizer cost functions for different values of M, but not to reoptimize the query, which could result in different execution plans. The optimization of dynamic query execution plans is considered in [CoG94].

The next two sub-sections discuss our use of ROC for the allocation of memory and disk bandwidth, respectively. We also discuss initial resource requirements and the range over which allocations may vary, which gives the broker its ability to adapt.

4.2.1. Currency for Memory Allocation

The prototype memory broker uses ROC to guide memory allocations. Figure 2 illustrates the ROC curves for three different sizes of joins, where the maximum memory that may be allocated to any single operator is $M_{maxOp} = 512$ *pages* (one half of the total memory of 1024 pages in this example system). The results presented in [Yu93] showed the importance of distinguishing between "small" and "large" queries for memory allocation, and M_{maxOp} serves as our dividing point. In [Dav95], we show for quite different workloads that 50% of memory is a reasonable dividing point. A detailed discussion of ROC curves is given in [Yu93]. The basic idea is to give an operator the memory allocation that achieves the highest ROC, but to give the broker flexibility by allowing the memory allocation to vary. We refer to the ROC at $M_{\min} + 1$ as ROC_{base}, the baseline ROC (notice that ROC at $M_{\min}$ is undefined). The curves have a hook shape, and the bottom part of the hook is an undesirable allocation since it provides a lower ROC for a higher memory consumption than ROC_{base}. A join's memory allocation may range from its minimum requirement of $M_{\min}$ to the smaller of its maximum requirement of $M_{\max}$ and the maximum allocation made to any operator, M_{maxOp}. We refer to this maximum allocation as the maximum effective allocation, M_{maxEff}, and it is shown for each join in Figure 2. The initial memory allocation M_{init} is the memory allocation that results in the highest ROC over the range $M_{\min}$ to M_{maxEff}. For example, Join A, which is the smallest join, will initially receive its maximum allocation of 154 pages, while the larger Join B will receive 512 pages. The largest join is Join C, for which the highest ROC is ROC_{base}, resulting in an initial allocation of its minimum requirement of 31 pages. We give the broker flexibility to adapt to changes in the workload by allowing the allocations to vary after the initial allocation. To

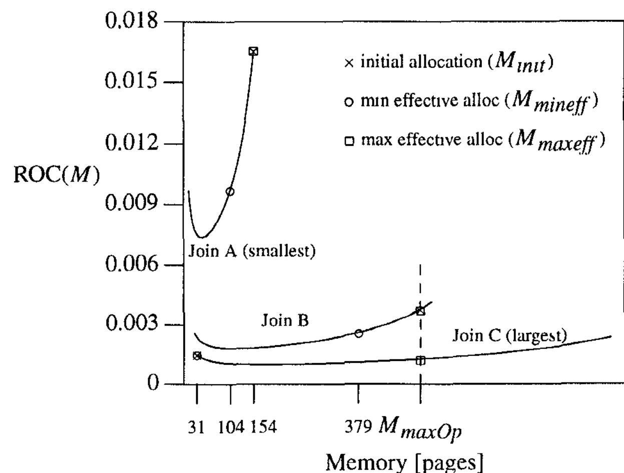

Figure 2. Return on Consumption.

avoid the undesirable allocations that result in lower ROC than ROC_{base}, we determine the smallest memory allocation larger than $M_{\min} + 1$ that provides higher ROC than ROC_{base}. This is called the minimum effective allocation M_{minEff}, and the broker may not reduce the join's allocation below this amount. Thus, a join's memory allocation may range from M_{minEff} to M_{maxEff}. For example, Join B may have its allocation reduced from 512 pages to as low as 379 pages. It is not possible to achieve a higher ROC for large joins, such as Join C. However, if memory is available that cannot be used by a higher bidder, it is preferable to gain some benefit by allocating the unused memory to this large join rather than letting the memory remain idle. As soon as there is a higher bidder that can use the memory, the large join will be required to sell excess pages back to the broker. Notice that although $M_{\min}$ may be a reasonable allocation for the smaller joins (Join A and Join B), we excluded it. We plan to study the effect of considering this allocation in the future. Allowing the allocations to vary gives the broker flexibility to adapt to changes in contention by "recalling" memory, and to fully exploit memory by deriving every possible benefit from idle memory.

The bid price, which guides the dynamic changes in allocations, is the highest ROC that the operator can achieve over the range of allowable memory allocations, which is the ROC at the join's initial memory allocation of M_{init} (or $M_{init} + 1$ for large joins). To determine the bid price for hash join, it is only necessary to reevaluate optimizer cost functions for $M_{\min} + 1$ and M_{maxEff}. This bid price achieves the desired effect that smaller joins submit higher bids than larger joins.

4.2.2. Currency for Disk Bandwidth Allocation

To allocate disk bandwidth, we reserve a portion of memory to sell to the bidders for use as I/O buffers. This will improve disk utilization by increasing I/O parallelism (the adaptable algorithm already exploits large block I/O by using dynamically sized I/O buffers, as discussed in [DaG94]). Operators that are allocated multiple buffers may use asynchronous I/O to increase their share of disk bandwidth. The buffers are used for asynchronous input (prefetching), but the operator may also use asynchronous output (see the upcoming sub-section "Adaptable Algorithms"). Intuitively, one might expect that more disk bandwidth should be allocated to small joins in preference over large joins since this was most effective for memory allocation. In fact, ROC for disk bandwidth indicates that a balanced approach is appropriate, with preference for *large* joins. Although large joins have preference, the difference between large and small joins is fairly small.

To determine how to allocate both memory and disk bandwidth at the same time, we consider the nature of the resources. Memory is a passive resource, meaning that it does not actually perform work for the join. The size of the join's memory allocation does, however, dictate the amount of work that must be done, i.e., the amount of intermediate I/O to partition overflow files. Once the amount of work is determined, active resources, such as disk and processor, perform the work. Since memory dictates the amount of work, it is the critical resource and should be determined first. We use a two-step approach to calculate ROC for disk bandwidth. First, we determine the initial memory allocation M_{init} as described in the previous sub-section. This dictates how much work must be done for the operator, if the allocation were to remain fixed. Second, using M_{init} as a constant, we determine ROC as a function of disk bandwidth only:

$$ROC(D) = \frac{T(M_{init}, D_{\min}) - T(M_{init}, D)}{D \times T(M_{init}, D) - D_{\min} \times T(M_{init}, D_{\min})}.$$

The operator is guaranteed to receive its initial allocation of M_{init}, but any changes in that allocation are difficult to predict since they are workload-dependent. Given the uncertainty of the workload, it is reasonable to use the allocation of M_{init} to determine the Return on Consumption for disk bandwidth. When calculating $T(M, D)$, we divided the I/O time by the number of I/O buffers, so that doubling the number of I/O buffers halves the I/O time. However, in our simulation, we observed sublinear speedup with additional disk buffers. This phenomenon is due to the manner in which our adaptable hash join algorithm exploits memory, and is discussed below. We account for this phenomenon in our calculation of $T(M, D)$ by including an "effectiveness factor" to reduce expected speedup (based on our observations, we used an effectiveness factor of 0.70). The detailed analysis for $T(M, D)$ was excluded due to space constraints, and may be found in [Dav95]. Figure 3 illustrates the ROC curves for disk bandwidth for the same three joins illustrated in Figure 2. The system has *numDisks* = 6 disks, and each join is limited to one buffer per disk. That is, the disk bandwidth for every operator may range from $D_{\min} = 1$ buffer (synchronous I/O) to $D_{\max} = numDisks$ buffers. We may make two observations from Figure 3. First, larger joins have a higher disk ROC than smaller joins. The reason is that large joins realize an enormous benefit from a small number of pages when the pages are used for disk bandwidth. This is in contrast to memory allocation, which requires a large increase in memory pages to achieve a substantial benefit, resulting in a large increase in cost. Second, the highest ROC is obtained for an allocation of $D = 4$ buffers, and larger allocations actually result in a lower ROC because they provide much smaller benefit than earlier buffers. Based on these observations, we define D_{peak} as the allocation of disk bandwidth that provides the highest disk ROC. The broker attempts to allocate D_{peak} buffers to each join, but allows allocations to vary from $D_{\min}$ to $D_{\max}$. If there is insufficient bandwidth to allocate D_{peak} buffers to all joins, lower bidders (smaller joins) must sell to allow higher bidders (larger joins) to retain D_{peak}. If all bidders have D_{peak} buffers, higher bidders are permitted to buy idle bandwidth in the range of $D_{peak} + 1$ to $D_{\max}$. Even though this reduces the disk ROC, it allows some benefit to be gained from resources that are not otherwise in use.

In summary, our heuristic for allocation of disk bandwidth is to attempt to allocate D_{peak} buffers to each join to achieve the highest disk ROC for all joins, but to give preference to larger joins if there is an insufficient number of buffers for every bidder.

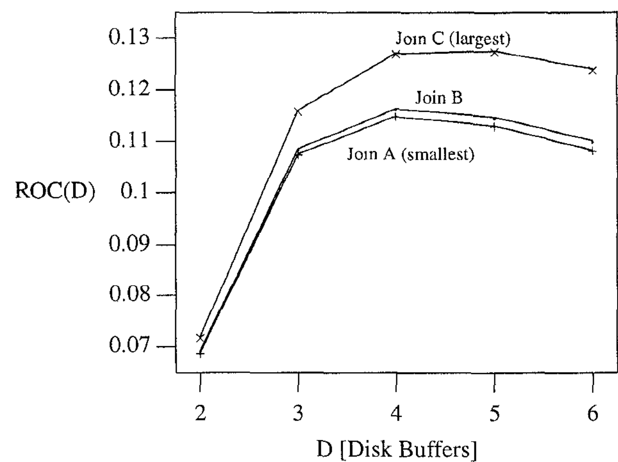

Figure 3. Return on Consumption for Disk Bandwidth.

Furthermore, large joins are permitted to buy idle buffers if all joins have D_{peak} buffers. The bid price for disk bandwidth is the highest disk ROC over the range of allowed allocations, creating the desired effect that large joins bid more for disk bandwidth than smaller joins. However, given the small performance difference between small and large joins, it would be reasonable to adopt a simpler allocation policy for disk bandwidth, such as equal allocation.

4.3. Admission Policy

To reduce response time and meet the constraint of fairness, we use an out-of-order admission policy which we call *slowdown-based admission*. For queries that are waiting to execute, slowdown is a measure of the effect of wait time on the query. Specifically,

$$slowdown = (waitTime + execTime_{est}) / execTime_{est}.$$

The query with the highest slowdown is the one to be considered for admission. Slowdown-based admission favors small queries but avoids starvation. Since slowdown increases over time, the slowdown for a large query will eventually become high, preventing starvation of large queries. In a heavily loaded system, biased wait-time may be a large contributor to unfairness. Slowdown-based admission reduces this bias and improves fairness by causing queries to wait in proportion to their estimated execution times.

Admission of a query is considered when a new query arrives and at each bid point. When admission is considered, the query with the highest slowdown becomes the next candidate for admission. Note that in the case of a newly arrived query, the new query may or may not be the candidate query. Once the candidate query is chosen, it may be admitted if the available and total reclaimable resources are sufficient to satisfy its needs. Available resources are those residing in the broker's short-term reserves. Memory allocated to an operator in excess of M_{minEff} is reclaimable, and an allocation of disk bandwidth in excess of one buffer is reclaimable. Once the query is admitted, the operators in the new query may be required to wait for resources, until lower bidders issue bids and are required by the broker to sell into the short-term reserves. When the available resources in the short-term reserves are sufficient, the operators receive their initial resource allocation (M_{init} pages of memory and at least one I/O buffer) and begin execution.

Since it could be very expensive to continuously maintain slowdown values for very many queries in a heavily loaded system, we divide the new queries into several queues based on estimated execution time $execTime_{est}$. Only the queries at the heads of the queues are considered for admission, thus only those queries must have their slowdown values updated. For our prototype broker, we use 10 queues. Since small queries are most vulnerable to excessive wait time, more queues should be dedicated to smaller queries.

The multiprogramming level (MPL) is not defined explicitly, but dynamically varies as necessary to maximize profit. Since memory is the primary resource and the allowed range of allocations for a query depends on its size, the workload will affect the MPL. A workload with many large queries, which have small minimum effective allocations (M_{minEff}) will have a higher MPL than a workload with mostly small queries, which have larger minimum effective allocations. This is precisely the reason that it is important for the resource manager to manage disk bandwidth in addition to memory.

4.4. Allocation Policy

We discussed the initial resource allocations and the range over which the allocations may vary in Section 4.2. In this sub-section, we describe the policies that control these dynamic resource fluctuations. Resource allocations are driven by demand, and are guided by the currency. We adopted a simple rule for buying and selling resources: only the highest bidder may buy, and only the lowest bidder may sell. This stabilizes the system by avoiding chain reactions, such as a high-price bidder buying from a mid-price bidder, which then buys from a low-price bidder. The broker manages two lists per resource: *demandList*, which contains bidders that have less than their M_{maxEff}, and *excessList*, which contains bidders that have more than their M_{minEff}. Both lists are ordered by bid price, which is inexpensive since bid prices are fixed and bidders are inserted and removed from the lists very infrequently.

The broker reacts to rather than anticipates changes in resource contention, necessitating that operators interact with the broker on a fairly frequent basis. Furthermore, our prototype broker is passive since it is implemented as a shared memory data structure, making it necessary for the active operators to "poll" the broker. This polling also ensures that the operator is at a point where it may quickly alter its resource usage. Although there is frequent interaction, bidding is very inexpensive since it is trivial to find the highest bidder that demands additional resources and the lowest bidder that has excess resources. To give the broker control and to allow it to react within a reasonable time frame, operators are required to bid every *bidInterval* seconds. In addition, a join bids before spilling a partition (in hopes of gaining memory), and before each build overflow file is processed (the memory requirement may be larger or smaller than for the previous one). The *bidInterval* must be short enough to allow the broker to be responsive but not so short that bidding costs are excessive. In [Dav95], we show that bid overhead is very low (about .01% of CPU time).

4.5. Adaptable Algorithms

In Section 2, we briefly reviewed the MCRHJ adaptable hash join algorithm that we use in this study. That algorithm can dynamically adapt to fluctuations in its memory allocation, but as described in [DaG94], it uses synchronous I/O. Since the broker allocates both memory and disk bandwidth, we modified the join algorithm to dynamically adapt to fluctuations in its disk bandwidth allocation by using asynchronous I/O. When allocated multiple input buffers, the join is permitted to use both asynchronous input and output. Asynchronous input, or prefetching, is straightforward to include by simply issuing multiple reads. The join accomplishes asynchronous output by writing sooner than necessary. That is, the previous algorithm reacts by flushing or spilling a partition when the free list is exhausted. The modified algorithm reacts when the free list reaches a certain small size. For the purposes of this study, that size is *numDisks*, the number of disks in the system. This gives the join "slack time" so it can continue processing after issuing each asynchronous write request. Notice that all input attributed to the join uses the allocated input buffers for asynchronous input. When the join is consuming the original build or probe input, the scan operator uses the input buffers allocated to the join. However, if the input is pipelined from another operator, that operator may use only the resources allocated to it.

Asynchronous I/O will significantly reduce I/O time, but it will not provide linear speedup due to the tradeoff between dynamic, large cluster I/O and asynchronous I/O. Although asynchronous I/O is effective for scans of base tables, it results in smaller cluster sizes for intermediate writes and reads because the join algorithm

writes sooner than necessary. Since files are read with the same cluster size as they are written, this impacts all intermediate I/O.

5. Database Simulator

We implemented a detailed database simulator to study resource allocation techniques. The simulator uses the Awesime simulation toolkit [Gru91] and consists of a query source, CPU, resource broker, and disks. In this section, we describe the simulated hardware and software architectures, the workload, and the implementation of the policies against which we compare the broker.

Description	Parameter	Value
CPU Speed	*cpuSpeed*	50 MIPS
Memory	*M*	8 MB
Page Size	*pageSize*	8 KB
Number of Disks	*numDisks*	6
Disk Transfer Rate	*transferTime*	3.5 MB/sec
Disk Rotational Latency	*rotationalDelay*	4.76 ms
# Cylinders per Disk	*numCylinders*	2368
Disk Cylinder Size	*cylinderSize*	69 pages
Disk Seek Factor	*seekFactor*	247 μs

Table 1. Simulated Hardware Architecture.

5.1. Hardware and Software Architecture

Table 1 summarizes the simulated machine architecture, with the disk parameters taken from a Maxtor MXT-1240S. The elevator algorithm is used for I/O request scheduling. The disk access time is *diskAccess* = *seekTime* + *rotationalDelay* + *transferTime*. The time to seek across t tracks is calculated as *seekTime* = *seekFactor* $\times \sqrt{t}$ [BiG88]. The memory size of 8 MB may seem small, but was chosen intentionally to limit simulation time. The scalability of the broker is demonstrated in [Dav95].

The simulator provides file scan and hash join operators, and each query is a two-way join with two file scans. Each probe tuple matches with exactly one build tuple. Since resource allocation of multiple resources in a complex multiple-query workload is not well understood, we include only two-way joins so we can focus solely on resource allocation issues. Once we understand the problem for simple queries, we plan to extend our techniques to complex queries with pipelining and parallelism. The number of CPU instructions for each simulator operation is shown in Table 2. The broker is implemented as a shared memory data structure rather than a separate process. All permanent and temporary files are partitioned over all disks. Synchronous I/O is used for all policies except for the broker that manages both memory and disk bandwidth. Values of parameters relating to the software architecture are shown in Table 3.

Operation	#Instructions
Initiate a join	40,000
Terminate a join	10,000
Read a tuple from a memory page	300
Copy a tuple to output buffer	100
Insert a tuple into hash table	100
Hash a tuple	500
Probe the hash table	200
Start an I/O operation	2000
Complete an I/O operation	500
Make bid to broker	1000

Table 2. Number of CPU Instructions per Operation.

5.2. Workload

In the experimental section, we refer to small, medium, and large users, which repeatedly generate queries of the size designated. Each user randomly chooses its build and probe file from a group of files that are sized relative to the memory M in the system. The tuple size for all files is 256 bytes. Small build files range from 5% of M to 25% of M, and small probe files range from 25% of M to M. Medium build files range from 30% of M to M, and medium probe files range from M to $4 \times M$. Large build files range from M to $4 \times M$, and large probe files range from $4 \times M$ to $16 \times M$. Before issuing a query, a user thinks for an amount of time that follows a negative exponential distribution with a mean as follows: small user 3 seconds; medium user 45 seconds; large user 240 seconds.

5.3. Comparison Policies

To the best of our knowledge, few studies have addressed the problem of resource allocation in a multi-user environment [BMC94, MeD93a, PCL94, Yu93]. Of these studies, two are not relevant to our objective because they address completely different problems. Although a dynamic memory allocation technique was proposed in [PCL94], that technique has a performance objective which is specific to real-time systems. Complex transaction and query workloads with per-class response time goals were studied in [BMC94]. In our experiments, we compare to the two remaining policies, *ROC* and *Class*, described below. The inferiority of static allocation techniques has been shown [Yu93], as has the inferiority of proportional allocation techniques [PCL94, Yu93]. Therefore, we do not include these techniques, allowing us to concentrate on those techniques that have previously demonstrated the best performance with respect to our objective.

ROC is the technique presented by Yu and Cornell with the goal of minimizing average response time [Yu93]. Given a target ROC, operators with lower ROC receive their minimum memory allocation while operators with higher ROC receive their maximum memory allocation. Once an operator receives its memory allocation, the allocation remains fixed for the duration of its execution. *ROC* uses FCFS admission.

Class is the technique proposed by Mehta and DeWitt with the goal of fairness [MeD93a], described in Section 2. Although *Class* does adapt by dynamically changing the per-class MPLs, the memory allocations remain fixed for the duration of an

Simulator Software Architecture		
Description	Parameter	Value
Hash table fudge factor	*fudge*	1.2
Input buffer size	C_{input}	8 pages
Target cluster size	C_{tgt}	8 pages
Min effective cluster	C_{eff}	4 pages
Bid interval	*bidInterval*	5 sec
Max memory per op	M_{maxop}	.50 of M
Memory for disk BW	M_{diskBW}	24 buffers, 192 pg
Best disk BW alloc	D_{peak}	4 buffers, 32 pg

Table 3. Software Architecture.

operator's execution. Based on the recommendations in [MeD93b] and on our own experiments, we set the boundary between small and medium queries at 25% of memory, and the medium-large boundary at 95%, to achieve the best performance. Note that with hash table overhead, a "small" build file of 25% of M is classified as medium.

6. Experimental Results

We graph the performance of each policy as the number of users in the system increases (i.e., as the system load increases). Memory is the scarce resource, and there is sufficient disk bandwidth for all policies. We report the following metrics as appropriate to compare and contrast the behavior of the different policies: slowdown, fairness, CPU and disk utilization, the size of I/O requests, and the number of resource fluctuations for the broker policies. Slowdown and fairness, the primary metrics, were described in detail in Section 4.1. Lower values are preferred for both of these metrics. Slowdown indicates how much higher is the response time in a multi-user environment compared to a single-user environment; a lower value means a better response time was obtained. Fairness indicates how evenly performance degraded among all the queries; a lower value means that the degradation was applied fairly among all queries. For the primary metrics, slowdown and fairness, the 90% confidence interval is within a few percent of the mean. Unless otherwise noted, the experiments were run for 3 hours of simulated time. $Broker_M$ is the broker policy that allocates only memory, and $Broker_{MD}$ allocates both memory and disk bandwidth. Recall that *ROC*, *Class*, and $Broker_M$ all use synchronous I/O, and $Broker_{MD}$ uses asynchronous I/O.

6.1. Mostly Small Users

In this experiment, we show the performance of the policies for a workload that contains a majority of small users. Taking think times into account, the actual ratio of concurrent small:medium:large users is approximately 4:1:1. Figure 4 shows the slowdown and fairness for this experiment as the number of users in the system increases to produce a heavier workload. The techniques give similar performance with few users in the system, but as the system load increases the techniques diverge in performance. The broker techniques provide the best and most stable performance for both slowdown and fairness as the number of users increases. We will first discuss *ROC* and *Class*, which show the highest variability, and then we will discuss $Broker_M$ and $Broker_{MD}$.

The performance of *ROC* at the lightest loads is the closest to $Broker_M$, but it worsens significantly as the number of users increases. The allocation policy of *ROC* is effective, but its FCFS admission policy results in poor performance under heavier system loads. As the system load increases, *ROC* allows more large queries to execute with small memory allocations. These queries have long execution times, so small queries are penalized with high wait times. In addition to increasing overall slowdown, this admission policy harms fairness. The inadequacy of FCFS admission for the goal of fairness has been noted before [MeD93a].

Class uses a more sophisticated admission policy that produces lower slowdowns than *ROC* at higher loads, but still much higher than the broker. Surprisingly, *Class* gives the worst fairness. First, we examine the effect of the *Class* admission policy on performance. This policy improves performance compared to the strict FCFS ordering of *ROC*. However, variability in query size within each class still causes smaller queries to suffer extra waiting time behind larger queries, particularly as the number of users increases. The second factor that affects the performance of *Class* is its memory allocation policy. *Class* uses an equal allocation policy for medium queries. This policy is ineffective to reduce response time, and combined with the dynamic per-class MPLs, it results in unfairness.

The broker policies provide the best performance and stability as the system load increases. CPU utilization for this experiment is shown in Figure 5, and disk utilization is shown in Figure 6. $Broker_M$ generally has lower disk utilization than *ROC* and *Class*, and higher CPU utilization. Notice that the higher CPU utilization is not due to brokering overhead, but is a result of the effectiveness of $Broker_M$ in utilizing the disk devices. To explain this, we must examine the effectiveness of the underlying hash join algorithm. Recall that the MCRHJ hash join algorithm exploits memory by dynamically varying the size of the I/O buffer, or cluster. Ample memory allows the algorithm to use a larger cluster, while memory scarcity causes a smaller cluster. Thus, the allocation policy directly affects this aspect of the join's performance. Figure 7 shows the average I/O cluster size realized by each technique. $Broker_M$ typically achieves the highest cluster size of all algorithms. This improved effectiveness in using the disk device decreases disk utilization and improves CPU utilization. As the system load increases, *ROC* uses smaller, less effective cluster sizes, resulting in an increase in disk utilization and a corresponding decrease in CPU utilization. Since *ROC* uses

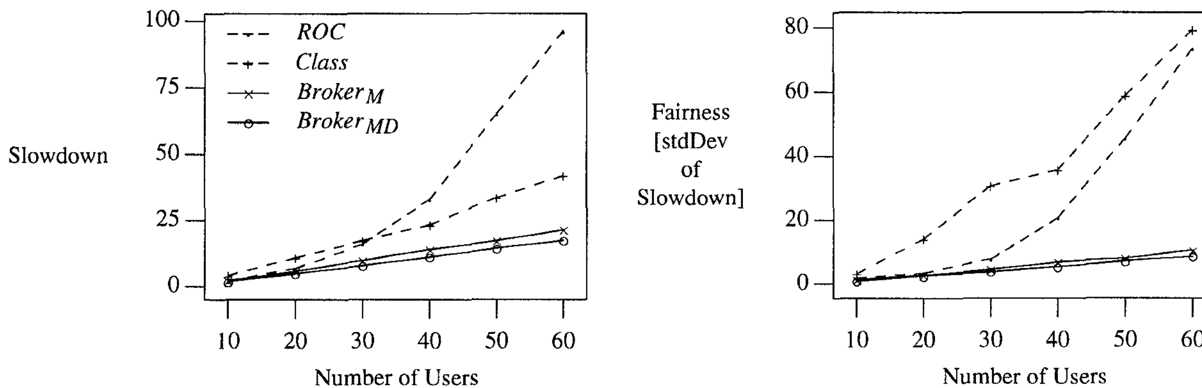

Figure 4. Mostly Small Users (Primary Metrics).

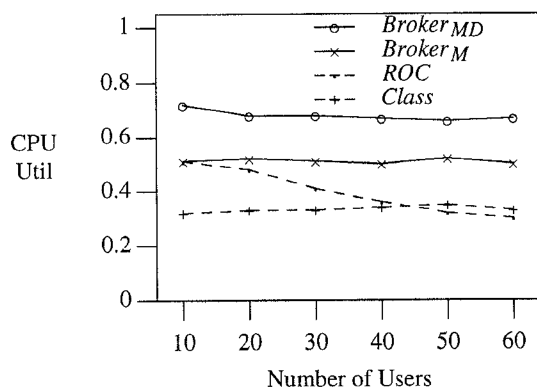

Figure 5. CPU Util for Mostly Small Users.

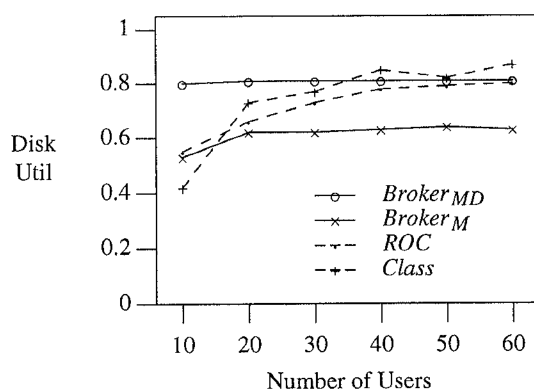

Figure 6. Disk Util for Mostly Small Users.

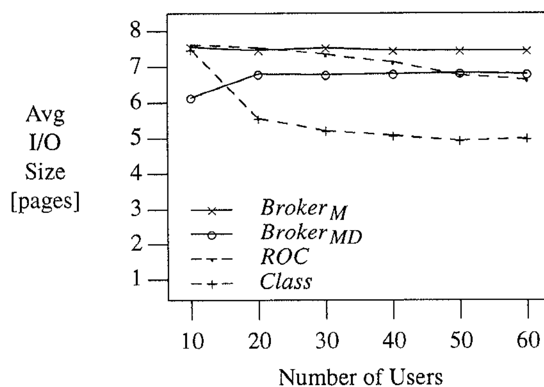

Figure 7. Average I/O Cluster Size for Mostly Small Users.

FCFS admission, a heavier load means more large queries will be active due to their longer execution times. Large queries execute with their minimum memory allocation, and thus use a small cluster size. *Class* also uses a smaller cluster size with increasing system load, which increases disk utilization. The smaller cluster size for *Class* is due to a larger medium MPL, which causes medium queries to receive smaller memory allocations. CPU utilization for *Class* remains stable because the policy increases the MPL with increasing load (this also contributes to the higher disk utilization). The low disk utilization of $Broker_M$ indicates that additional performance might be obtained if the utilization could be improved. Rather than increasing the MPL, $Broker_{MD}$ extends $Broker_M$ by allocating disk bandwidth in addition to memory to improve performance and device utilization. $Broker_{MD}$ explicitly allocates portions of disk bandwidth to individual queries by giving I/O buffers that the queries use for asynchronous I/O. By using asynchronous I/O, $Broker_{MD}$ improves both slowdown and fairness. $Broker_{MD}$ increases disk utilization to a desirable level and as a result, also improves CPU utilization, resulting in more balanced device utilizations.

The broker allocation policy is dynamic, and imposes resource fluctuations on the adaptable MCRHJ algorithm. Queries under $Broker_M$ average 2.16 fluctuations in their memory allocation, while $Broker_{MD}$ queries average 2.36 memory fluctuations and 1.4 disk bandwidth fluctuations. The average number of fluctuations is quite low, and is stable with increasing system load.

In summary, this experiment has shown that the broker is very effective in meeting the performance objective for a workload composed of mostly small users. Furthermore, performance is stable. The techniques performed similarly at light loads, but quite differently as the number of users increased. *ROC* gave poor performance at heavier system loads due to its FCFS admission policy. *Class* improved slowdown compared to *ROC*, but it still allowed extra wait time for the smaller queries in each class. The *Class* equal memory allocation policy for medium queries resulted in unfairness. $Broker_M$ was very effective in meeting the objective for this workload, but it exhibited lower disk utilization than *ROC* and *Class*. $Broker_{MD}$ was able to achieve an improvement in performance compared to $Broker_M$ by allocating disk bandwidth, improving both disk and CPU utilization.

6.2. Mostly Large Users

In this experiment, we changed the ratio of concurrent queries (small:medium:large) to 1:1:4, and left everything else the same as the previous experiment. Figure 8 shows the slowdown and fairness for this workload as the number of users in the system increases. The relative results are very similar to the previous experiment with mostly small users, so we will only discuss the *Class* technique, which exhibits noticeably different results.

Class shows improvement in both slowdown and fairness compared to its results in the previous experiment. The low number of small queries prevents a significant wait-time bias among queries in the small class. Queries in the large class have a very low memory requirement, so many of them may be executed concurrently resulting in lower wait times. Furthermore, longer wait times can be more easily absorbed by large queries without significantly affecting their slowdown or fairness. The improvement in fairness is due to the smaller number of medium queries. Recall that the largest "small" build file was classified by *Class* as medium. Thus, reducing the number of small users reduces the number of *Class* medium queries. This limits the unfairness against medium queries that occurred in the previous experiment.

The broker policies minimize the number of memory fluctuations for smaller joins since their memory allocation may not vary much from maximum. Large joins, on the other hand, may experience more fluctuations since they are allowed to buy idle memory for the periods of time it is not required by higher bidders (smaller joins). For this workload, $Broker_M$ queries average 8.12 fluctuations in their memory allocation compared to 2.16 for the previous workload. $Broker_{MD}$ queries average 7.3 memory fluctuations and 1.46 disk bandwidth fluctuations compared to 2.36 and 1.4, respectively, for the previous workload. This is still quite low, considering the longer average response

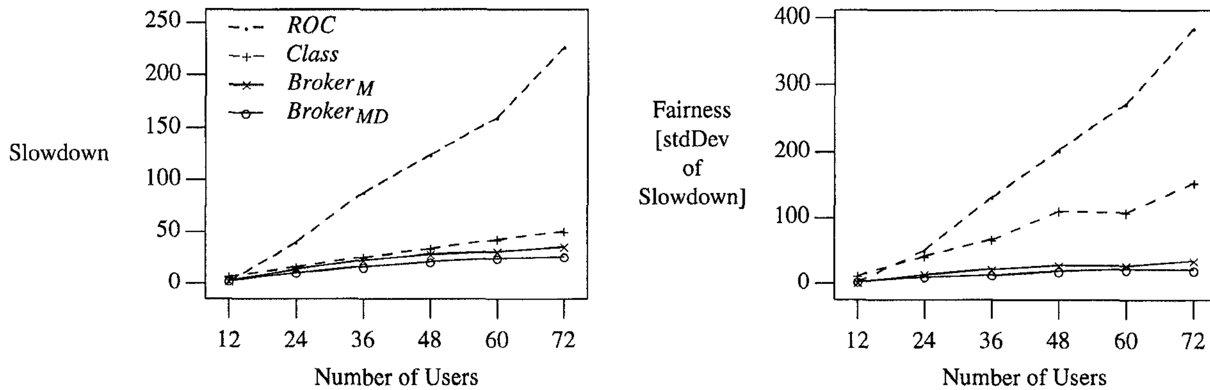

Figure 8. Mostly Large Users (Primary Metrics).

times for this workload.

6.3. Small and Large Users

In this experiment, we consider a workload that contains an even number of small and large users (small, medium, large ratio of 1:0:1). We have shown how the techniques handle workloads dominated by both small and large users. Since large queries have the greatest capacity to "interfere" with small queries, we explore the ability of the techniques to balance the demands of large queries against an equal number of small queries. The slowdown and fairness for this workload are shown in Figure 9. $Broker_M$ and $Broker_{MD}$ again provide excellent and stable performance for both slowdown and fairness. *ROC* provides poor performance as the number of users increases due to its FCFS admission policy as previously discussed. *Class* also provides poor performance for reasons discussed in Section 6.1. The difference in *Class* performance compared to that in Section 6.1 is that performance worsens significantly as the number of users increases from 12 to 48, but is fairly stable from 48 to 72 users. This experiment contains a large number of medium queries (the larger small queries that are categorized as medium queries by *Class*). *Class* favors these queries by increasing the medium MPL, but the medium MPL cannot be increased beyond the point at which medium queries receive their minimum allocation. This saturation point is reached at approximately 48 users, so performance won't worsen as significantly with the additional users.

7. Summary and Conclusions

In this paper, we have studied the problem of resource allocation for multi-user query execution. The research in this paper makes two main contributions. First, we proposed a new framework for dynamic resource allocation based on concepts from microeconomics. Second, we developed a prototype broker that shows the viability of the framework and contributes to the important goal of reducing response time in a multi-user environment.

The framework uses concepts from microeconomics to simplify the complex problem of allocating multiple resources among many competing queries. In order to execute, operators buy resources from a resource broker. The currency is derived from the performance objective, which allows the broker to obtain the best performance by simply maximizing its profitability. Moreover, the broker employs dynamic techniques and adapts to changes in resource contention by changing previous allocation decisions. This requires that the system include resource-adaptable algorithms, such as [DaG94, PCL93a]. The broker is logically a single entity, but may be implemented as a collection of entities to address a range of resource allocation problems, including architecture- or workload-specific problems.

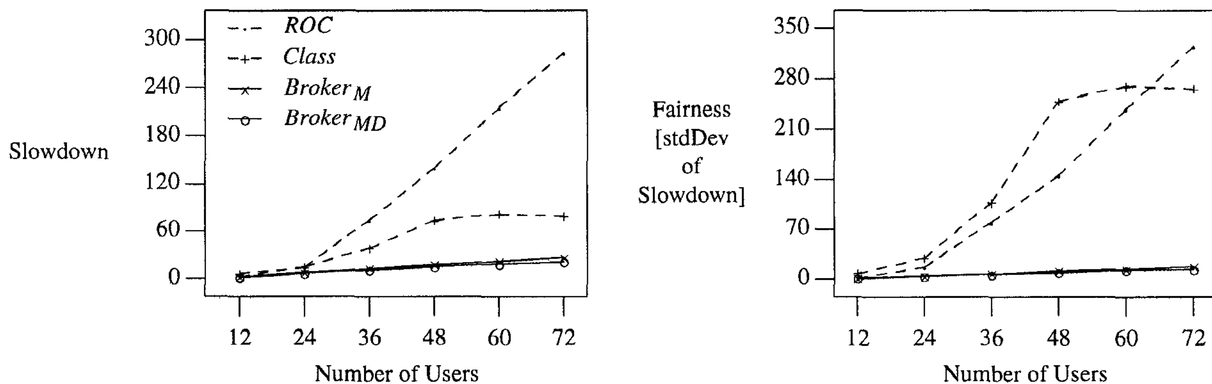

Figure 9. Even Small and Large Users (Primary Metrics).

We developed a prototype memory allocation broker ($Broker_M$) to show the viability of the framework. The performance objective of this prototype is to minimize slowdown with the constraint of fairness. We compared the memory broker to the best techniques to-date for minimizing response time and for obtaining the best fairness. $Broker_M$ outperformed the other two techniques by providing both better slowdowns and better fairness for a variety of workloads. This demonstrated the usefulness of the framework and of the policies in the prototype. However, the other techniques achieved higher disk utilization than the memory broker. We then extended the broker to allocate disk bandwidth as well as memory ($Broker_{MD}$). To allocate disk bandwidth, we used a portion of memory as I/O buffers for asynchronous I/O. We used a two-step approach for the allocation of both resources, where the best memory allocation is determined and then the allocation of disk bandwidth is determined. Whereas the heuristic for memory allocation is to give more to small joins, we showed that a balanced approach with a preference for large joins is better for disk bandwidth. However, the rather small difference between large and small joins indicates that a simple policy such as equal allocation would be effective for disk bandwidth allocation. By allocating both resources, $Broker_{MD}$ was able to obtain a reasonable performance improvement in both slowdown and fairness, compared to $Broker_M$. $Broker_{MD}$ was able to achieve much higher device utilizations, and more balanced disk and CPU utilizations than all the other algorithms. This was due to the techniques of large cluster I/O and asynchronous I/O. Large cluster I/O decreases I/O cost, reducing disk utilization and improving CPU utilization. Asynchronous I/O also reduces I/O cost and improves CPU utilization, but it improves disk utilization. In contrast to our approach, earlier work improves device utilization by increasing the multiprogramming level, which decreases the amount of memory available for each query. Instead, our results showed that it is better to use the best memory allocation since it dictates the amount of work that will be done. Then disk bandwidth can be allocated to improve both device utilization and performance.

Acknowledgements

Bob Devine and the anonymous reviewers made excellent suggestions for the presentation of this paper, which we appreciate very much.

References

[BiG88] D. Bitton and J. Gray, Disk Shadowing, *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 331.

[BMC94] K. P. Brown, M. Mehta, M. J. Carey, and M. Livny, Towards Automated Performance Tuning For Complex Workloads, *in Proc. Int'l. Conf. on Very Large Data Bases*, Santiago, Chile, September 1994, 72.

[CLL85] M. J. Carey, M. Livny, and H. Lu, Dynamic Task Allocation in a Distributed Database System, in *Proc. 5th Int'l. Conf. in Distr. Computing Sys.*, IEEE Computer Society, 1985, 282.

[CoG94] R. L. Cole and G. Graefe, Optimization of Dynamic Query Execution Plans, *Proc. ACM SIGMOD Conf.*, Minneapolis, MN, May 1994, 150.

[DaG94] D. L. Davison and G. Graefe, Memory-Contention Responsive Hash Joins, *Proc. Int'l. Conf. on Very Large Data Bases*, Santiago, Chile, September 1994, 379.

[Dav95] D. L. Davison, Dynamic Resource Allocation for Multi-User Query Execution, *Ph.D. Thesis, Univ. of Colorado at Boulder*, 1995. In preparation.

[DGS90] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, The Gamma Database Machine Project, *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 44.

[FNS91] C. Faloutsos, R. Ng, and T. Sellis, Predictive Load Control for Flexible Buffer Allocation, *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991, 265.

[FYN88] D. Ferguson, Y. Yemini, and C. Nikolaou, Microeconomic Algorithms for Load Balancing in Distributed Computer Systems, *Proc. 8th IEEE Int'l. Conf. on Distr. Computing Sys.*, San Jose, CA, June 1988.

[GlL89] F. R. Glahe and D. R. Lee, *Microeconomics Theory and Applications*, Harcourt Brace Jovanovich, 1989.

[GrD93] G. Graefe and D. L. Davison, Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Processing, *IEEE Trans. on Softw. Eng. 19*, 8 (August 1993), 749.

[Gru91] D. Grunwald, A Users Guide to AWESIME: An Object Oriented Parallel Programming and Simulation System, *Univ. of Colorado Tech. Rep. Univ. of Colorado at Boulder-Comp. Sci.-552-91*, Boulder, CO, November 1991.

[MeD93a] M. Mehta and D. J. DeWitt, Dynamic Memory Allocation for Multiple-Query Workloads, *Proc. Int'l. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993, 354.

[MeD93b] M. Mehta and D. DeWitt, Dynamic Memory Allocation for Multiple-Query Workloads, *Univ. of Wisconsin – Madison Comp. Sci. Tech. Rep. 1151*, 1993.

[NF91] R. Ng, C. Faloutsos, and T. Sellis, Flexible Buffer Allocation Based on Marginal Gains, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 387.

[PCL93a] H. Pang, M. J. Carey, and M. Livny, Memory-Adaptive External Sorting, *Proc. Int'l. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993, 618.

[PCL93b] H. Pang, M. J. Carey, and M. Livny, Partially Preemptible Hash Joins, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 59.

[PCL94] H. Pang, M. J. Carey, and M. Livny, Managing Memory for Real-Time Queries, *Proc. ACM SIGMOD Conf.*, Minneapolis, MN, May 1994, 221.

[SDK94] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin, An Economic Paradigm for Query Processing and Data Migration in Mariposa, *Third Int'l. Conf. on Parallel and Distr. Inf. Systems*, Austin, TX, September 1994.

[WHH92] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, Spawn: A Distributed Computational Economy, *IEEE Trans. on Softw. Eng. 18*, 2 (February 1992), 103.

[Yu93] P. S. Yu and D. W. Cornell, Buffer Management Based on Return on Consumption in a Multi-Query Environment, *The VLDB Journal 2*, 1 (January 1993), 1.