Multiclass Query Scheduling in Real-Time Database Systems

HweeHwa Pang, Michael J. Carey, *Member, IEEE*, and Miron Livny

***Abstract*—In recent years, a demand for real-time systems that can manipulate large amounts of shared data has led to the emergence of real-time database systems (RTDBS) as a research area. This paper focuses on the problem of scheduling queries in RTDBSs. We introduce and evaluate a new algorithm called *Priority Adaptation Query Resource Scheduling* (PAQRS) for handling both single class and multiclass query workloads. The performance objective of the algorithm is to minimize the number of missed deadlines, while at the same time ensuring that any deadline misses are scattered across the different classes according to an administratively-defined miss distribution. This objective is achieved by dynamically adapting the system's admission, memory allocation, and priority assignment policies according to its current resource configuration and workload characteristics. A series of experiments confirms that PAQRS is very effective for real-time query scheduling.**

***Index Terms*—Query processing, real-time database systems, memory management, priority scheduling.**

I. Introduction

A NUMBER of emerging database applications, including aircraft control, stock trading, network management, and factory automation, have to manipulate vast quantities of shared data in a timely manner. More specifically, these applications may generate transactions and queries that have to be completed by certain deadlines for the results to be of full (or perhaps even any) value [1], [26], [23]. The need for systems that are able to support such timely management of substantial amounts of data has sparked researchers' interest in the area of real-time database systems (RTDBS) in both the database and real-time computing communities. Most work in the RTDBS area to date has focused on the issues of transaction management and low-level resource (CPU and I/O) scheduling.

Depending on the extent to which its applications can tolerate violations of their time constraints, a real-time database system can be characterized as being either *hard*, *soft*, or *firm* [23]. In this study, we will focus on firm RTDBSs, where a job is considered useless once its deadline has passed [12]. In order to meet the time constraints of its jobs, a firm RTDBS must employ multiprogramming so that all of its resources can be utilized productively. Moreover, it must regulate the completion time of individual jobs; to do so, it uses priority scheduling to resolve any resource contention that stems from multiprogramming.

The performance objective of a firm RTDBS becomes even more demanding when its workload contains jobs that are drawn from a number of distinct classes. For such workloads, the RTDBS must also deal with the issue of how to distribute any deadline misses across the different classes in the workload. Since the desired distribution of misses may vary from one environment to another, the RTDBS should be able to tailor its resource scheduling policies based on a distribution provided by the system administrator. Thus, the objective of an RTDBS with a multiclass workload should be to minimize the total number of missed deadlines subject to the constraint that any misses be distributed as per the administrator's specification. In this paper, we will focus on the challenges associated with meeting this objective for multiclass, query-oriented, firm RTDBS workloads.

A. Real-Time Query Processing

The performance of queries can vary dramatically depending upon the amount of memory that they are given to work with. When given enough memory, most queries can simply read their operand relations once and produce the desired results directly. This amount is referred to as the queries' maximum memory requirements. Given less memory, as long as the amount given exceeds the queries' minimum memory requirements, most queries can still run by writing out temporary files and then reading them back in for further processing. For instance, a hash join can either execute with its maximum required memory, which is slightly greater than its inner relation size, or it can run in only one additional pass with as few buffer pages as the square root of its inner relation size [25]. In order to help all query classes attain their desired level of performance, it may be necessary for an RTDBS to increase concurrency by admitting some queries with less than their maximum memory allocations, particularly when queries have large memory requirements. If too many queries are admitted, however, the resulting extra I/Os could lead to thrashing, making high concurrency harmful instead of helpful. RTDBSs must therefore carefully control query admissions into the system.

Having determined which queries to admit, the next issue that an RTDBS faces is memory allocation. While the highest-priority query at a given CPU or disk will use that resource exclusively, memory must be shared among all admitted queries. When the aggregate maximum memory requirement of the admitted queries exceeds the available memory, the RTDBS must decide on the amount of memory to give each

Manuscript received Oct. 3, 1994; revised Mar. 24, 1995.

H.-H. Pang is with the Institute of Systems Science, National University of Singapore, Heng Mui Keng Terrace, Kent Ridge, Singapore 0511, Republic of Singapore; e-mail: hhpang@iss.nus.sg.

M.J. Carey is with IBM Almaden Research Center, 650 Harry Road, K55-B1, San Jose, CA 95120-6099; e-mail: carey@almaden.ibm.com.

M. Livny is with the Computer Sciences Department, University of Wisconsin at Madison, 1210 West Dayton Street, Madison WI 53706, USA; e-mail: miron@cs.wisc.edu.

IEEECS Log Number K95046.

query. This decision needs to take into account both the classes' performance requirements and the tightness of each query's timing constraint. In addition, the effectiveness of memory allocation in reducing individual queries' response times should be considered so as to make the best use of the available memory [8], [27]. Finally, since the relative priority of an executing query may change over time as other queries enter and leave the system, the memory allocation of a query is likely to fluctuate. To facilitate efficient query processing in the face of such memory fluctuations, RTDBSs require query operators that can dynamically release memory [8], [28] as well as accept more memory [20], [21], [9] while they are executing. To date, the admission control and memory allocation issues that arise in real-time query scheduling have not been addressed.

B. Our Focus

This paper focuses on the problem of scheduling queries in real-time database systems. We propose and evaluate an algorithm called *Priority Adaptation Query Resource Scheduling* (PAQRS, pronounced "packers") that is designed to schedule both single- and multiclass query workloads. This algorithm provides mechanisms to dynamically adapt the admission control and memory allocation decisions of an RTDBS to the system resource configuration and the characteristics of the workload. Moreover, PAQRS is equipped with a class-sensitive bias control mechanism. When presented with a heavy multiclass workload, this mechanism exercises explicit control over the relative priority of the individual classes, thus regulating their performance to conform to the administratively defined miss distribution.

The remainder of this paper is organized as follows: Section II briefly discusses related work. Section III reviews an algorithm, called PMM [22], that provides the basis for the development of the PAQRS algorithm. PAQRS itself is presented in Section IV. Section V describes a detailed simulator of a firm RTDBS that will be used to study the performance of the PAQRS algorithm. Section VI presents the results of a series of experiments showing that, over a wide range of workloads, PAQRS offers an effective solution to the query scheduling problem in RTDBSs. Finally, our conclusions are presented in Section VII.

II. Related Work

A significant body of work exists in the real-time database system area [23], but almost all of this work has focused on issues and algorithms related to either real-time transaction scheduling (e.g., [1], [12], [14], [15]) or real-time disk scheduling [2], [3], [7], [15]. To the best of our knowledge, the problem of scheduling queries in an RTDBS has not been addressed to date. As a result, the only studies that are closely related to the work reported here are two studies that have examined resource scheduling for multiclass query workloads in the context of traditional (non-real-time) database systems.

In [8], [27], Cornell and Yu introduced the concepts of memory consumption and return on consumption (ROC) as the basis for memory management in a multiquery environment. Using these concepts to characterize the effect of memory allocations on query response times, a heuristic algorithm was proposed to allocate memory among concurrently running queries in a way that ensures a certain level of ROC. An important result from this study is that giving some of the queries their maximum required memory, while allocating the minimum possible memory to the rest, leads to near-optimal memory usage. This result is directly incorporated in the memory allocation strategies of PAQRS.

The problem of meeting predefined performance objectives in a multiclass database system was recently studied in [6]. In that study, Brown et al. explored the problem of automatically adjusting the multiprogramming levels (MPL) and memory allocations of a database management system to achieve per-class response time goals for multiclass workloads. An algorithm called M&M was introduced to find MPL and memory settings for each class; these settings are determined dynamically by a feedback mechanism that is driven by a set of heuristics and estimation techniques. Simulation results showed that M&M can successfully achieve response times that are within a few percent of the goals. Despite its promise, M&M cannot be directly used in the RTDBS context. This is because, being priority-ignorant, M&M may choose MPL and memory settings that conflict with the job priorities that are used for concurrency control and the scheduling of the CPU and disks. Thus, a complete solution, one that integrates priority assignment, MPL control, and memory allocation, must be sought.

III. Basic Real-Time Query Scheduling

In a firm real-time database system, a query becomes worthless if it fails to complete by its deadline. The primary performance objective of an RTDBS is, if possible, to meet all query deadlines. If this is not possible, and if all queries are of equal importance, then the RTDBS will try to minimize the number of missed deadlines. In [22], we presented a query scheduling algorithm based on this performance objective. The algorithm, called Priority Memory Management (PMM), regulates memory usage for firm real-time query workloads. Since PAQRS builds on this algorithm, we describe PMM in detail in this section before introducing the PAQRS algorithm in Section IV.

The PMM algorithm consists of an admission control component and a memory allocation component. Both components employ the Earliest Deadline (ED) scheduling policy [16], so queries that are more urgent are given higher priority in admission and memory allocation decisions than queries whose deadlines are further away. (We adopt the ED policy here, as opposed to policies that take into account query execution times, because reasonable execution time predictions are not usually available in multiuser database systems.) The admission control component of PMM sets the target multiprogramming level (MPL) by statistical projection from past miss ratios and their associated MPL values. In cases where this method fails, PMM falls back on a heuristic that chooses the

MPL based on desirable resource utilization levels. The memory allocation component operates using one of two strategies —a Max strategy that assigns to each query either its maximum required memory or no memory at all, and a MinMax strategy that allows some low-priority queries to run with their minimum required memory while the high-priority ones get their maximum. The current choice of memory allocation strategy is based on statistics about the workload characteristics that PMM gathers. Since both the MPL setting and memory allocation strategy choices have to be tailored to the characteristics of the workload, PMM constantly monitors the workload for changes that may necessitate adjustments to its decisions. The details of the algorithm are presented below. The key parameters of PMM, which will be explained as they appear in the following description, are summarized in Table I.

TABLE I
PMM ALGORITHM PARAMETERS

Parameter	Meaning	Default
SampleSize	Reevaluation frequency (number of query completions)	30
$[Util_{Low}, Util_{High}]$	Range of "desirable" CPU/ disk utilization levels	[0.70, 0.85]
$Adapt_{ConfLevel}$	Conf. level of statistical tests for PMM adaptation	95%
$Change_{ConfLevel}$	Conf. level of statistical tests for workload changes	99%

A. Admission Control

The task of the admission control mechanism is to determine the MPL based on current operating conditions. In order to minimize the *miss ratio*, defined as the proportion of queries that fail to complete by their deadlines, the MPL has to be high enough so that the CPU and disk resources can be fully exploited. However, the MPL should not be so high as to cause the system to experience thrashing. The relationship between MPL and miss ratio thus follows the shape of a concave curve. PMM attempts to locate the optimal MPL, i.e., the MPL that leads to the lowest miss ratio on this curve, through a combination of *miss ratio projection* and a *resource utilization heuristic*, revising its MPL setting after every *SampleSize* queries are served by the system. The two components of the MPL determination method are presented below.

A.1 Miss Ratio Projection

The miss ratio projection method approximates the relationship between MPL and miss ratio by a concave quadratic equation; this equation is used to set the system's target MPL. A quadratic equation is used here because it stabilizes faster than higher-order equations, while still capturing the general shape of the concave curve. After every *SampleSize* query completions, PMM measures the miss ratio, $miss_i$, that the current MPL, mpl_i, produces. Based on this pair of values, together with past miss ratios and their associated MPL settings, a new quadratic equation is calculated according to the least squares method [11]. It is important to note that PMM does not actually have to keep track of individual miss ratio readings, but only the values of k, Σmpl_i, Σmpl_i^2, Σmpl_i^3, Σmpl_i^4, $\Sigma miss_i$, $\Sigma mpl_i \times miss_i$, and $\Sigma mpl_i^2 \times miss_i$, where k is the number of times PMM is invoked. Therefore, the space overhead incurred by the projection method is very low. The computation overhead is also minimal since the method requires only that the above summations be updated after every query completion, and deriving the quadratic equation entails only simple arithmetic involving these summations. After approximating the equation, a new MPL value is chosen according to the type of curve obtained:

Type 1. *The curve has a bowl shape*. In this case, the curve has a minimum. Therefore, the target MPL is set to the minimum of the curve. (This is the expected case after the algorithm has been operating for a while.)

Type 2. *The curve is monotonically decreasing*, i.e., higher MPLs lead to lower miss ratios. This indicates that the optimal MPL is beyond the highest MPL tried so far. Since the curve may not be valid if extrapolated too far, the projection method selects an MPL that is one above this largest attempted MPL. Next, PMM applies the resource utilization heuristic (described below) to see if an even higher MPL may be warranted. If so, the MPL suggested by that heuristic is adopted; otherwise PMM maintains the MPL that the miss ratio projection method picked.

Type 3. *The curve is monotonically increasing*. The MPL computation procedure for this case is just the opposite of the procedure for Type 2 curves. Here the projection method tentatively selects an MPL that is one unit below the smallest MPL that has been tried so far. Next, a second MPL is obtained using the resource utilization heuristic. The two MPLs are then compared, and the smaller of the two is adopted.

Type 4. *The curve has a hill shape*. Occasionally the fitted curve takes on this shape due to randomness in the observed miss ratios caused by inherent workload fluctuations. When this happens, the projection method fails and PMM resorts to the resource utilization heuristic.

An attractive feature of the miss ratio projection method is that the MPL values that it picks improve over time: Initially, the shape of the fitted curve is largely influenced by random workload fluctuations. As time progresses and more miss ratio readings are obtained, the fitted curve will gradually stabilize and its optimum will close in on the optimal MPL. At this point, the system can be expected to deliver good performance so long as there are no significant changes in the workload characteristics. (Workload changes will be addressed in Section III.C).

A.2 Resource Utilization Heuristic

The resource utilization (RU) heuristic attempts to help the system achieve low query miss ratios by keeping the utilization of the most heavily loaded resource among the CPUs and disks within some "desirable" range, $[Util_{Low}, Util_{High}]$, thus avoiding situations where the bottleneck resource is either underutilized or near saturation. The heuristic extrapolates from the current MPL and utilization to predict a new MPL that is likely to bring the utilization into the middle of the $[Util_{Low}, Util_{High}]$ range by applying the following formula:

$$MPL_{New} = \frac{Util_{Low} + Util_{High}}{2 \times Util_{Current}} \times MPL_{Current}$$

The linear dependency between MPL and utilization that this formula assumes is based on the observation that the utilization of a resource increases approximately linearly with the MPL until the resource is near saturation, at which point the utilization levels off. Since neither the RU heuristic nor the miss ratio projection method are likely to push the utilization way above $Util_{High}$ to the saturation point, the above formula should provide satisfactory MPL estimates most of the time. Even in regions where the linear dependency assumption does not hold, the RU heuristic is still useful in steering the MPL setting in the direction of the optimal MPL since utilization increases monotonically with MPL.

As described, a value that the RU heuristic uses to compute the new MPL is the utilization of the most heavily loaded resource at the current MPL. Due to random workload fluctuations, the utilization over the duration of the current batch of *SampleSize* queries may not be indicative of the resource's overall average utilization at that MPL. For this reason, the heuristic actually averages the utilization values that have been obtained so far instead of relying only on the most recent utilization reading. Conceptually, PMM computes the average utilization at the current MPL, denoted as $Util_{Current}$ in the formula above, by first obtaining a straight line from every pair $\langle util_i, mpl_i \rangle$ of observed utilization values and their associated MPLs by using the least squares method [11], again applying the linearity assumption. The average utilization is then taken from the fitted line as the rate that corresponds to the current MPL. For the purposes of computing the straight line, PMM records the values of k, Σmpl_i, Σmpl_i^2, $\Sigma util_i$, and $\Sigma mpl_i \times util_i$, where k denotes the number of times PMM is invoked. As discussed earlier, the space and computation overheads involved are minimal.

B. Memory Allocation

As described above, queries like hash joins and external sorts each have a maximum and a minimum memory requirement. Given its maximum required memory, such an operation can read its operand relation(s) and generate results directly. Given only its minimum required memory, which is typically much lower than its maximum, the operation instead has to process its operand relation(s), write out intermediate results to temporary files, and then read these files back for further processing before the final results can be produced. The maximum memory requirement of an external sort is the size of its operand relation [25], whereas it can run with as few as three memory pages by doing multiple merge passes. In the case of a hash join, the maximum memory requirement and the "minimum" memory demand (for two-pass operation) are $F\|R\|$ and $\sqrt{F\|R\|}$, respectively, where $\|R\|$ is the inner (building) relation size and F is a fudge factor that reflects the overhead of a hash table [25].

When the total maximum memory requirement of the admitted queries exceeds the available memory, the memory allocation component is responsible for determining the amount of memory to allot to each query. As mentioned previously, the memory allocation decisions are based on the ED policy, so queries that are more urgent are always given buffers ahead of queries with looser deadlines. At any given time, PMM adopts one of two memory allocation strategies: the Max strategy or the MinMax policy. With the Max strategy, queries are either allocated enough memory to satisfy their maximum demands or else they are given no buffers at all. When operating in MinMax mode, however, PMM is able to admit more queries by meeting the maximum memory demands for only some of the more urgent queries, allowing the rest of the queries to execute with their minimum required memory. The reason for doing MinMax allocation, as opposed to simply dividing the available memory proportionally among the admitted queries, is that MinMax leads to more effective use of memory than proportional allocation (as was shown in [28], [27], [6] for non-real-time database systems and in [22] for RTDBSs). A possible concern about MinMax is that it may allow too many queries to run with minimum memory allocations, thereby overloading the disks. However, this situation does not arise with PAQRS because the number of queries that are eligible for memory allocation is regulated by the admission control component of PAQRS.

The MinMax allocation process is conceptually carried out in two passes. Starting from the highest-priority query, PMM first gives each query just enough memory for it to begin execution. If there are leftover buffers at the end of this pass, PMM makes another pass through the list of admitted queries, again beginning with the highest-priority query. In the second pass, the allocation of each query in turn is topped up to its maximum. The allocation process terminates when either all of the available memory has been allocated or all of the queries have received their maximum allocations. At the end of this memory allocation process, one possible scenario is that some of the higher-priority queries will have their minimum allocations, while the lower-priorities are suspended due to a shortage of memory. Another scenario is that the higher-priority queries will have their maximum allocations while the lower-priority queries just have their minimum. The only possible exception is the query that gets the last few memory pages in the second pass, which may receive an allocation somewhere in between its minimum and maximum demands. In a running system, of course, queries do not arrive all at once; rather, they come and go over time. Since the ED policy assigns priorities to queries according to their urgency, the memory allocation of a query can therefore vary between maximum, minimum, or no memory allocation as higher-priority queries enter and leave the system, but over time it will settle on the maximum allocation as the query's deadline draws close. In order to deal with these fluctuations in the query's memory allocation, PMM has to rely on adaptive query processing operators, e.g., adaptive joins and sorts, to adjust the query's memory usage dynamically.

The Max strategy, by insisting on the maximum memory allocation, eliminates the potential thrashing problem that can result when additional (lower-priority) queries are admitted at the expense of requiring some of the higher-priority queries to run with less than their maximum memory allocations. Consequently, PMM does not need to explicitly limit the MPL when it is in Max mode. Instead, PMM admits as many higher-priority queries—at their maximum allocations—as the available memory permits. A possible pitfall of Max is that it may severely restrict the MPL if every query requires a substantial portion of the system memory in order to run at its maximum allocation. In contrast to Max, MinMax assigns to some or all of the admitted queries as little as their minimum memory demand, thus enabling the system to achieve the target MPL that the admission control component sets. Whether Max or MinMax performs better depends on the workload characteristics and the system configuration—Max is preferable if memory is abundant and the bottleneck resource type is CPU or disk, whereas MinMax is more suitable for memory-constrained situations.

The PMM algorithm uses a feedback mechanism to monitor the state of the system, and it revises its choice of allocation strategy as necessary. Initially, the Max mode is selected. After serving every *SampleSize* queries, PMM checks the system state and switches to the MinMax strategy if all of the following conditions are met: 1) one or more queries in this batch missed their deadlines; 2) the utilizations of all CPUs and disks are below $Util_{Low}$, which indicates that none of these resources are likely to be a bottleneck; 3) there is a nonzero admission waiting time, suggesting that there is memory contention; and 4) on the average, the execution time of a query is shorter than its time constraint (the difference between its deadline and its arrival time) so that the longer execution times that will result from switching to the MinMax strategy are likely to be feasible. In checking for condition 3), PMM carries out a large-sample test [10] for the mean waiting time at a confidence level of $Adapt_{ConfLevel}$. Condition 4) is tested in a similar fashion, except that here the test is performed on the difference between the execution time and time constraint. After switching to MinMax, PMM then monitors the target MPL. If the target MPL setting drops to or falls below the average MPL that was realized in Max mode, PMM reverts to the Max strategy. This entire process is repeated continuously.

C. Dealing with Workload Changes

PMM attempts to minimize query miss ratios by tailoring its MPL and memory settings to the system's workload and resource configuration. Consequently, it is necessary for PMM to discard the statistics that it has gathered and to re-adapt itself when the workload undergoes a significant change. In order to detect workload changes, PMM constantly monitors the following workload characteristics: 1) the average maximum memory demand of queries; 2) the average number of I/Os that each query issues to read its operand relation(s);[1] and 3) the average normalized time constraint, defined as the ratio of the time constraint to the number of I/Os needed to read the operand relation(s). After every *SampleSize* query completions, PMM carries out a large-sample test with a confidence level of $Change_{ConfLevel}$ [10] on each monitored workload characteristic to see if its present value differs significantly from its last observed value. If so, PMM concludes that a workload change has taken place. Since every workload change prompts PMM to restart itself, $Change_{ConfLevel}$ is set to a high value (see Table I) to reduce the chances of PMM wrongly reacting to inherent workload fluctuations.

1. The number of I/Os that are expended to write and read intermediate results depends on memory allocation decisions, and thus is not an inherent characteristic of the workload.

D. An Example

Having presented the PMM algorithm in detail, we now finish by illustrating it with a simple example of the algorithm's operation. Suppose that the first batch of *SampleSize* queries produces point *a* in Fig. 1a under the Max strategy, and suppose that PMM concludes that Max is inappropriate and decides to switch to MinMax. At this point, the RU heuristic suggests a higher MPL, from which we derive point *b* after the next batch of query completions. Once more, the RU heuristic leads PMM to raise its MPL setting, which results in point *c* after the third batch of queries. Having collected three observations, PMM can now apply the miss ratio projection method. The quadratic equation that is computed from the three points is shown by the Type 2 curve (see Section III.A.1) in Fig. 1a. This curve causes PMM to experiment with an even higher MPL, the consequence of which is indicated by point *d* in Fig. 1b. Applying the projection method again, PMM now obtains a Type 1 curve. Since the optimum of the curve is likely to be near the optimal point, PMM adopts the MPL value associated with this optimum for its next MPL setting. As this process continues and more observations are gathered, the fitted curve will gradually stabilize and lead PMM to the best MPL for the given workload.

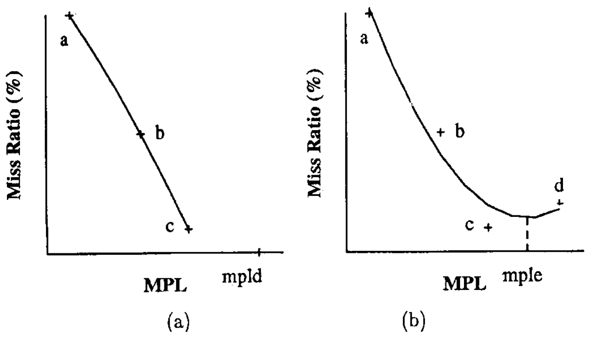


Fig. 1. Admission control decision making.

IV. Multicalss Real-Time Query Scheduling

As discussed earlier, it may be desirable in some environments to distribute missed deadlines proportionally among query classes according to administratively-defined workload objectives. To address the need for such controlled performance, this section presents *Priority Adaptation Query Resource Scheduling* (PAQRS), an algorithm for scheduling multiclass firm real-time query workloads. Given such a workload, PAQRS allows a system administrator to specify a list of values,

$$RelMissRatio = \{relMissRatio_1 : \ldots : relMissRatio_{NumClasses}\},$$

that indicates the desired miss ratio distribution among the classes in the workload. For example, suppose that the workload is made up of two classes. If *RelMissRatio* = {3:1}, then the target miss ratio distribution would be of the form $MissRatio_1$ = 3x% and $MissRatio_2$ = 1x% for some x. The performance objective of PAQRS is to minimize the number of missed deadlines, subject to the constraint that miss ratios must be distributed among the classes in the manner specified by the system administrator. The PAQRS algorithm's multiprogramming level (MPL) control and memory allocation mechanisms are based on those of PMM. In addition, a bias control mechanism is provided to allow PAQRS to intervene on behalf of classes that require help in order to meet the performance objective. The details of the algorithm are presented below. Its input parameters and variables, which will be explained as they appear in the following description, are summarized in Table II.

TABLE II
NOTATION FOR PAQRS

Parameter	Meaning	Default
RelMissRatio	Target relative class miss ratios	{1 : ... : 1}
$SampleSize_{Class}$	Reevaluation frequency (no. queries per class)	10
$SampleSize_{Total}$	Reevaluation frequency (total no. queries)	30
$Change_{ConfLevel}$	Conf. level of statistical tests for workload changes	99%

Variable	Meaning	Default
$RegQuota_i$	Class *i*'s quota of regular queries	–
$MissRatio_i$	Measured miss ratio of class *i*	–
$Weight_i$	Weight of class *i* in weighted miss ratio	from *RelMissRatio*
P_Q	Priority of query *Q*	–
D_Q	Deadline of query *Q*	–
R_Q	Random key assigned to query *Q*	[0, 1]

A. Overview of the PAQRS Algorithm

Like PMM, PAQRS sets a system-wide target MPL and a global memory allocation scheme (Max or MinMax). Unlike PMM, which strives only to minimize the overall number of missed deadlines, PAQRS chooses its MPL and memory allocation strategy based on a class-sensitive performance measure. Moreover, class-sensitive triggers are used to determine when workload changes necessitate revisions to those choices. PAQRS is thus able to make admission and memory allocation decisions that complement its bias control mechanism in helping all classes achieve their desired level of performance.

The bias control mechanism of PAQRS controls the performance of individual classes by regulating the priority of their queries. Roughly speaking, PAQRS accomplishes this regulation using a multiclass variant of the Adaptive Earliest Deadline scheduling policy proposed in [13]. PAQRS divides all queries into two priority groups—a *regular* group and a *reserve* group—and a quota of regular queries is chosen for each class of query. Priority values are assigned to regular queries based on the Earliest Deadline policy [16], while reserve queries are assigned random priorities that are lower than those of any regular query; regular queries are always admitted and allotted resources ahead of reserve queries. By raising the quota of regular queries for classes that would naturally miss more deadlines than desired, and by limiting the number of regular queries from classes that would otherwise tend to miss fewer deadlines, PAQRS is able to distribute missed deadlines among the query classes according to the specified workload objectives.

B. Admission Control and Memory Allocation in PAQRS

As mentioned above, the PAQRS algorithm adapts PMM to choose a system-wide target MPL and a global memory allocation strategy that are conducive to meeting the workload's multiclass performance objective. PAQRS does this by basing its MPL selection decisions on a system-wide performance measure that better reflects the desired miss ratio distribution, and by picking its memory allocation strategy according to the level of memory contention experienced by individual classes.

The primary mechanism that PAQRS relies on to pick its target MPL settings is a statistical projection method that predicts the MPL value that will lead to the lowest "average" miss ratio. Thus, we need an "average miss ratio" computation procedure that suitably reflects the desired influence of the individual classes. Intuitively, if we want

$$rel\ Miss\ Ratio_i = c \times rel\ Miss\ Ratio_j$$

for two classes *i* and *j*, then class *i* should exert *c* times as much influence as class *j* on the "average" miss ratio. This is achieved by first transforming the values in *Rel Miss Ratio* into class weights:

$$Weight_i = \frac{\frac{1}{rel\ Miss\ Ratio_i}}{\sum_j \frac{1}{rel\ Miss\ Ratio_j}}$$

and then computing a weighted miss ratio for the projection method from the individual classes' miss ratios and their corresponding weights:

$$Weighted\ Miss\ Ratio = \Sigma\ Weight_i \times Miss\ Ratio_i$$

To illustrate how this procedure works, let us again consider a two-class workload with *Rel Miss Ratio* = {3:1}. Applying the above procedure, the two classes would be assigned weights of 0.25 and 0.75, respectively, making class 2 three times as influential as class 1. An important property of the class weights is that they add up to 1.0. This property ensures that the weighted sum of the class miss ratios, each of which ranges from 0% to 100%, remains within the interval [0%, 100%].

Having adjusted the MPL selection mechanism, we now turn our attention to the way that PAQRS chooses its memory allocation strategy. To adapt better in a multiclass context, PAQRS needs to replace the system-wide performance measures that PMM uses with class-sensitive measures. PAQRS starts with the Max allocation strategy and then switches to MinMax mode if the utilization of all CPUs and disks are below $Util_{Low}$ and some class *i* satisfies all of the following conditions: 1) one or more queries from that class have missed their deadlines since PAQRS was last activated; 2) class *i* has a nonzero admission waiting time; and 3) on the average, the execution time of a query belonging to class *i* is significantly shorter than its time constraint (the difference between its deadline and its arrival time). In other words, PAQRS switches to MinMax mode if some class appears to be missing deadlines unnecessarily because its queries are made to wait for memory. Since the above tests require performance statistics for all of the classes, PAQRS is invoked to revise its choices of MPL and memory allocation strategy only after the system has served at least $SampleSize_{Class}$ queries from every class, in addition to the original requirement of $SampleSize_{Total}$ total query completions, subsequent to the PAQRS algorithm's last activation.

Finally, to ensure that its choices of MPL setting and memory allocation strategy remain suitable for the workload, PAQRS constantly monitors the following statistics for each class: 1) the average maximum memory demand of queries in that class; 2) the average number of I/Os that each query in that class issues to read its operand relation(s); and 3) the average normalized time constraint, defined as the ratio of the time constraint to the number of I/Os needed to read the operand relation(s), for that class. Upon activation, PAQRS carries out a *t*-test on each monitored class characteristic to see if its present value is different from its last observed value at a confidence level of $Change_{ConfLevel}$ [10]. If so, PAQRS reacts to the workload change by discarding the statistics that it has gathered and by readapting itself to the new workload composition. The differences between PAQRS and PMM thus far are summarized in Table III.

TABLE III
SUMMARY OF DIFFERENCES BETWEEN PMM AND PAQRS

	PMM	PAQRS
Miss ratio projection	$Avg\ Miss\ Ratio = \frac{\#late\ queries}{\#queries}$	WeightedMiss Ratio $= \Sigma\ Avg\ MissRatio_i \times Weight_i$
Memory allocation	Switch from Max to MinMax if queries in workload experience significant unnecessary memory waiting time	Switch from Max to MinMax if queries in some class experience significant unnecessary memory waiting time
Reactivation frequency	$SampleSize_{Total}$ queries	$\geq SampleSize_{Total}$ queries $\geq SampleSize_{Class}$ queries/class
Restart condition	Changes in average workload characteristics	Changes in the characteristics of some class *i*

C. Bias Control in PAQRS

While PAQRS picks its MPL and memory allocation strategy according to the target miss ratio distribution, this alone does not always suffice to ensure that the distribution is achieved. In many cases, an RTDBS can produce biased behaviors that do not conform to the requirements of its given multiclass objective. To rectify such undesirable behavior, PAQRS is equipped with a bias control mechanism that helps classes that would otherwise miss more deadlines to attain acceptable relative miss ratios by regulating the relative priorities of the classes.

As mentioned earlier, PAQRS divides queries into a regular group and a reserve group. Each class *i* is given a quota of regular queries, $RegQuota_i$, that limits the maximum number of regular queries that the class may have at any given time. Upon arrival, a query belonging to class *i* is assigned to the regular group if that class has not used up its quota of regular queries; otherwise the query is relegated to the reserve group of the class. Having determined the query's grouping, the following scheme is used to compute a two-part priority for the query:

$$P_Q = \begin{cases} 1, 1/D_Q & \text{if } Group = regular \\ 0, R_Q & \text{if } Group = reserve \end{cases}$$

where P_Q, D_Q, and R_Q denote, respectively, the query's priority, deadline, and a randomly assigned value in the range [0, 1]. This scheme defines a lexicographical priority order in which higher P_Q values reflect higher priorities. All regular queries have higher precedence than queries in the reserve group. Among queries in the regular group, priority rankings are established according to the ED policy. Priority ordering within the reserve group follows the Random Priority (RP) policy, which is why the R_Q values are selected randomly. The reason that RP is chosen for the reserve group is because its queries essentially "see" a heavily loaded system due to their lower priorities, and RP (unlike ED) delivers good performance under heavy loads [13].

PAQRS attempts to meet the target miss ratio distribution by elevating the priority of classes that suffer from higher-than-desired miss ratios, thus helping their queries to gain admission and compete for system resources. This is accomplished by increasing the regular query quota, $RegQuota_i$, of those disadvantaged classes, and by reducing $RegQuota_i$ for classes that are overachieving. At system start-up time, all $RegQuota_i$'s are first initialized to ∞ so that all queries are assigned to the regular group. When PAQRS is next activated, it first resets $RegQuota_i$ for each class to the highest number of concurrent queries that the class experienced during the intervening period and then adjusts the $RegQuota_i$'s according to the relative performance of the classes. If the target miss ratio distribution is achieved, all of the classes should bear an equal share of the weighted miss ratio. For example, if the target miss ratio distribution $RelMissRatio$ = {3:1} for a two-class workload is reached, the weighted miss ratio should be:

$$\begin{aligned} WeightedMissRatio &= Weight_1 \times MissRatio_1 + Weight_2 \\ &\quad \times MissRatio_2 \\ &= 0.25 \times 3x\% + 0.75 \times x\% \\ &= 0.75x\% + 0.75x\% \end{aligned}$$

In other words, $Weight_i \times MissRatio_i$ should be equal to $WeightedMissRatio/NumClasses$ for all classes *i*. If the current miss ratio distribution is different from the target, PAQRS adjusts the $RegQuota_i$ of each class based on how its $Weight_i \times MissRatio_i$ value compares to its share of $WeightedMissRatio/NumClasses$, using the following formula:

$$RegQuota_i^{new} = RegQuota_i^{old} \times \frac{Weight_i \times MissRatio_i}{WeightedMissRatio/NumClasses}$$

Returning to our $RelMissRatio$ = {3:1} example, if currently the miss ratio of classes 1 and 2 are 20% and 10%, respectively, PAQRS will reduce $RegQuota_1$ by 20% and increase $RegQuota_2$ by 20% in an attempt to bring the class miss ratios closer to the target distribution. After that, PAQRS continues to monitor the relative performance of the classes, applying the above formula to dynamically adapt the *RegQuota*$_i$s as needed.

The integration of the bias control and admission control mechanisms of PAQRS is straightforward—the m highest-priority queries get admitted, where m is the target MPL. However, the use of the two-tier priority scheme does introduce some difficulty in memory allocation. In particular, if MinMax mode is selected, should reserve queries be given their minimum required memory before the allocation of regular queries are topped up to their maximum, or should the memory manager start giving buffers to reserve queries only after all of the regular queries have received their maximum required memory? Since the purpose of the two-tier priority scheme is to help disadvantaged classes compete for system resources by relegating some queries from the advantaged classes to the reserve group, we adopt the second alternative to maximize the effectiveness of the scheme, i.e., reserve queries are not allowed to compete for memory with regular queries. To implement this alternative, we extend the MinMax allocation procedure of PMM to a two-step procedure. In the first step, the MinMax allocation procedure is applied to distribute memory to the regular queries; reserve queries are not eligible for allocation in this step. Step two, which uses MinMax to assign memory to the reserve queries, is activated only if there are leftover buffers at the end of step one (which only happens when all regular queries have been given their maximum required memory).

As noted earlier, the two-tier priority assignment scheme adopted by PAQRS follows the same concept as the Adaptive Earliest Deadline (AED) algorithm [13]. This algorithm was proposed to stabilize the overload performance of the ED scheduling policy. AED maintains a "hit" group and a "miss" group, which correspond to the regular group and reserve group in PAQRS, and AED controls "hit" group assignments by a *HitSize* parameter. The distinction between the two algorithms lies in the goals that they hope to reach with the two-tier scheme. In the case of ÀED, the single *HitSize* parameter serves to keep down the number of transactions that are scheduled according to the ED policy, whereas PAQRS uses its vector of *RegQuota*$_i$ values to influence relative class priorities. Consequently, the procedures that the algorithms employ to set their control parameters are quite different.

V. Database System Simulation Model

To aid in our study of real-time query scheduling issues, we have constructed a simulation model of a centralized database system. In subsequent sections, this simulation model will be used to evaluate the performance of PAQRS. The model, shown in Fig. 2, has five components: a *Source* that generates the workload of the system and collects statistics on completed queries; a *Query Manager* that models the execution details of queries, including hash joins, external sorts and sort-merge joins; a *Memory Manager* that implements an LRU replacement policy and the PMM and PAQRS algorithms; and a *CPU Manager* and a *Disk Manager* that are responsible for managing the system's CPU and disks, respectively. In this section, we describe how the simulation model captures the details of the database, workload, and various physical resources of a database system. We also summarize the algorithms employed by the Query Manager that enable executing queries to adapt to fluctuations in their memory allocations. The simulator is written in DeNet [17].

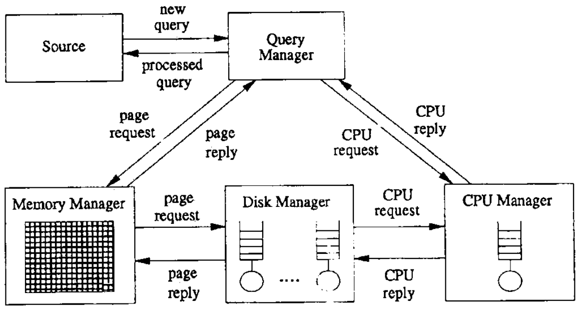


Fig. 2. Database system model.

A. Database and Workload Model

Table IV summarizes the database and workload parameters that are relevant to this study. Our objective for this study is to simulate a stream of external sorts and/or hash joins on different relations. To facilitate this, the database consists of *NumGroups* groups of relations. Each group i has *RelPerDisk*$_i$ clustered relations per disk. The size of the *RelPerDisk*$_i$ relations are chosen at equal intervals from *SizeRange*$_i$. For example, if *RelPerDisk*$_i$ = 5 and *SizeRange*$_i$ = [100, 200] pages, group i will have five relations with sizes equal to 100, 125, 150, 175, and 200 pages, respectively, on every single disk. To minimize disk head movement, all relations assigned to the same disk are randomly placed on its middle cylinders; temporary files are allotted either the inner cylinders or the outer cylinders.

TABLE IV
Database and Workload Model Parameters

Database	Meaning
NumGroups	Number of relation groups in the database
RelPerDisk$_i$	Number of relations per disk for group i
SizeRange$_i$	Range of relation sizes for group i
TupleSize	Tuple size of relations in bytes

Workload	Meaning
NumClasses	Number of classes in the workload
QueryType$_j$	Type of class j queries (hash join or external sort)
RelGroup$_j$	Operand relation group(s) for class j queries
λ_j	Arrival rate of class j queries
SRInterval$_j$	Range of slack ratios for class j queries
F	Fudge factor for hash joins

In this study, the workload comprises *NumClasses* classes of queries. Each class j has the following characteristics: It may be made up of external sorts, in which case *RelGroup*$_j$ specifies a group of database relations from which queries in class j draw their operand relations. Alternatively, the class may consist of hash joins. In the second case, every query in the class randomly chooses two relations by taking one relation from each of the two relation groups listed in

RelGroup$_j$. The smaller of the two chosen relations is the inner relation, **R**, of the join, while its outer relation, **S**, is the larger relation. The type of queries that form the class (sort or hash join) is indicated by the parameter *QueryType$_j$*. Query submissions from the class follow a Poisson process with a mean arrival rate of λ_j. The *Source* module assigns a deadline to each new query Q from class j in the following manner:

$$Deadline_Q = StandAlone_Q \times SlackRatio_Q + Arrival_Q$$

where $Deadline_Q$, $StandAlone_Q$, $SlackRatio_Q$ and $Arrival_Q$ are the deadline, standalone execution time, slack ratio and arrival time of query Q, respectively. The standalone execution time of a query is the time it would take to execute alone in the system, i.e., without experiencing any contention from other queries. The slack ratio, $SlackRatio_Q$, varies uniformly in the range specified by *SRInterval$_j$*, and it controls the tightness of the query's assigned deadline.

B. Physical Resource Model

The parameters that specify the physical resources of our model, which consist of a CPU, disks and main memory, are listed with their default values in Table V. An Earliest Deadline (ED) scheduling discipline is used for the CPU. The MIPS rating of the CPU is given by *CPUSpeed*. Table VI gives the costs of the various CPU operations involved in the execution of hash joins and external sorts.

TABLE V
PHYSICAL RESOURCE MODEL PARAMETERS

Parameter	Meaning	Default
CPUSpeed	MIPS rating of CPU	40 MIPS
NumDisks	Number of disks	10
SeekFactor	Seek factor of disk	0.000617
RotationTime	Time for one disk rotation	16.7 msec
NumCylinders	Number of cylinders per disk	1,500
CylinderSize	Number of pages per cylinder	90 pages
PageSize	Number of bytes per page	8 kbytes
BlockSize	Number of pages requested on each sequential I/O	6
M	Total number of buffer pages	2,560 pages

TABLE VI
NUMBER OF CPU INSTRUCTIONS PER OPERATION

Operation	# Instr	Operation	# Instr
Common Operations		**Hash Joins—**	
Start an I/O operation	1,000	*Hash tuple and insert into hash table*	100
Initiate a sort or join	40,000	*Hash tuple and probe hash table*	200
Terminate a sort or join	10,000	*Hash tuple and copy to output buffer*	100
		External Sorts—	
		Copy a tuple to output buffer	64
		Compare two keys	50

Turning to the disk model parameters in Table V, *NumDisks* specifies the number of disks attached to the system. Every disk manages its own queue by the ED scheduling policy; any disk requests that ED assigns the same priority to are serviced according to the elevator algorithm. Each disk has a 256-kbyte cache for use in prefetching pages. To keep the per-page I/O cost low, all queries capitalize on this facility, fetching *BlockSize* pages into the cache on each sequential I/O that incurs a disk cache miss (except during the merge phase of an external sort). Moreover, whenever queries have enough buffers, they spool their outputs so that pages are flushed to disk in blocks. The access characteristics of the disks are also given in Table V. Using the parameters in this table, the total time required to complete a disk access is:

$$Disk\ Access\ Time = Seek\ Time + Rotational\ Delay + Transfer\ Time$$

As in [4], the time required to seek across n tracks is:

$$Seek\ Time\ (n) = Seek\ Factor \times \sqrt{n}$$

Finally, the system has a total buffer pool size of **M** pages. A memory reservation mechanism allows query operators, including sorts and joins, to reserve buffers for use as workspaces. These reserved buffers are managed by the operators themselves, while page replacement for nonreserved buffers is handled according to the LRU policy.

C. Memory-Adaptive Query Primitives

In a priority scheduling environment such as an RTDBS, large queries involving operations like hash joins and external sorts face the prospect of having memory taken away and/or allocated to them during their course of execution. For this reason, the simulated Query Manager employs adaptive algorithms to help queries adjust efficiently to such memory fluctuations. While PAQRS is designed to work with any adaptive query processing operator, we will use the adaptive hash join and external sorting algorithms that we found to deliver the best performance among a range of alternatives that we investigated in a recent pair of studies [20], [21]. The two algorithms are briefly summarized here.

The hash join algorithm that the Query Manager employs was introduced in [20] as *Partially Preemptible Hash Join* (PPHJ) with *late contraction*, *expansion*, and *priority spooling*. PPHJ splits the pair of input relations into a set of partitions, as is done in traditional hash joins as well. At any one time during join execution using PPHJ, some of these partitions may be *expanded*, i.e., held in hash tables in memory, while others are *contracted*, i.e., resident on disk. When asked by the memory manager to free up buffers, PPHJ can do so by reducing the number of expanded partitions. Moreover, if extra memory becomes available while the outer (probing) relation is being split, PPHJ can expand contracted partitions so that outer relation tuples that hash to these partitions can be joined directly and then discarded, thus avoiding some I/Os.

The external sorting algorithm that is adopted in this study begins by using replacement selection to split the operand relation into sorted runs; these sorted runs are then repeatedly merged into longer runs until only a single run remains. These are the usual phases of an external sorting algorithm. What makes the algorithm adaptive is that, during the merging process, an executing merge step can be split into substeps that fit within the remaining memory if memory reductions occur [21]. Conversely, existing merge steps can be combined into larger steps (i.e., steps that merge more runs at once) to take advantage of any excess buffers that become available.

VI. Experiments and Results

This section presents the results of a series of experiments designed to evaluate the performance of the Priority Adaptation Query Resource Scheduling (PAQRS) algorithm. For comparison purposes, we shall also examine the original Priority Memory Management (PMM) algorithm, which does not distinguish between queries from different classes, i.e., which treats all queries like they belong to the same class. PMM is included here to highlight the ability of PAQRS to achieve targeted relative class performances, and also to reveal any price (in terms of system-wide performance metrics) that PAQRS may have to pay in the process.

We will begin our evaluation with an experiment where the workload consists of only one query class. This experiment is intended to give us an initial understanding of the admission and memory allocation mechanisms of PAQRS before we delve into the complexities introduced by multiple query classes. Since there is only one class here, PAQRS behaves exactly like PMM. We shall assess PAQRS by comparing it with two static algorithms: Max and MinMax. These two algorithms employ the Max policy and the MinMax strategy, respectively, in their memory allocation decisions. Next, we present a baseline experiment that is used to study the PAQRS algorithm's effectiveness in handling multiclass workloads. Further PAQRS experiments are then carried out by varying a few parameters each time. The primary performance metrics for these experiments are the *System Miss Ratio*, defined as

$$SystemMissRatio = \frac{Number\ of\ Late\ Queries}{Number\ of\ Submitted\ Queries},$$

the *Class Miss Ratio*, computed as

$$ClassMissRatio_i = \frac{Number\ of\ Late\ Queries\ in\ Class\ i}{Number\ of\ Class\ i\ Queries}$$

and the *WeightedMissRatio*. The weighted miss ratio combines the successes and failures of all classes into a single number that reflects how well the system performs as a whole, and is defined as

$$WeightedMissRatio = \Sigma\ Weight_i \times ClassMissRatio_i,$$

where $Weight_i$ is a weight assigned to class i according to an administratively defined performance objective. The way in which class weights are derived from the performance objective is described in Section IV.A. Unless stated otherwise, each experiment was run for 10 hours of simulated time, allowing a minimum of 2,000 completions per query class. We also verified that the size of the 90% confidence intervals for miss ratios (computed using the batch means approach [24]) was within a few percent of the mean in almost all cases.

A. Single Query Class

As mentioned above, our first experiment uses a single-class workload to study the MPL control and memory allocation mechanisms of PAQRS (and hence PMM). In order to bring out the importance of memory management, we simulate an environment where, except for occasional overloads, there are abundant CPU and disk capacities for the given workload; thus, memory is the bottleneck resource. This is achieved by letting *CPUSpeed* and *NumDisks* be 40 MIPS and 10, respectively, and by setting **M** to 2,560 pages (20 Mbytes). The rest of the resource parameters are kept at their settings of Table V The workload consists of one class of hash join queries. Each join has two operand relations, **R** and **S**, where $\|R\|$ varies uniformly between 600 and 1,800 pages and $\|S\|$ is selected from the range [3,000, 9,000] pages. Moreover, the slack ratio interval of this class is set to [2.5, 7.5]. The database and workload parameters are summarized in Table VII. A more comprehensive evaluation of PAQRS (PMM) involving various single-class workloads can be found in [22].

TABLE VII
Database and Workload Parameter Settings for the Single Query Class Experiment

Database	Meaning	Setting
NumGroups	Number of relation groups in the database	2
RelPerDisk$_1$	Number of relations per disk for group1	3
SizeRange$_1$	Range of relation sizes for group 1	[600, 1800] pages
RelPerDisk$_2$	Number of relations per disk for group 2	3
SizeRange$_2$	Range of relation sizes for group 2	[3000, 9000] pages
TupleSize	Tuple size of relations in bytes	256 bytes

Workload	Meaning	Setting
NumClasses	Number of classes in the workload	1
QueryType$_1$	Type of class 1 queries	Hash join
RelGroup$_1$	Operand relation groups for class 1 queries	1, 2
λ_1	Arrival rate of class 1 queries	varied (0.04 to 0.08)
SRInterval$_1$	Range of slack ratios for class 1 queries	[2.5, 7.5]
F	Fudge factor for hash joins	1.1

Fig. 3 plots the miss ratios for Max, MinMax, and PAQRS as a function of the arrival rate. The figure shows that MinMax consistently delivers the lowest miss ratio for this experiment, followed very closely by PAQRS. Max performs satisfactorily initially, achieving a near 0% miss ratio at λ = 0.04 queries/sec. As the arrival rate increases, however, the performance of Max deteriorates rapidly until, at λ = 0.08 queries/sec, Max produces a hefty 55% miss ratio, which is almost four times that of MinMax and PAQRS. These observations clearly show that the choice of memory allocation algorithm can have a very significant impact on the system miss ratio. To understand the behaviors of the three algorithms, we shall analyze each algorithm in turn with the aid of Figs. 4 and 5, which give the disk utilizations and average observed MPLs (as opposed to the target MPL set by PAQRS, which serves to limit the maximum MPL in the system), respectively.

Let us first examine the Max algorithm. This algorithm admits queries only if they can be allotted enough buffers to satisfy their maximum requirements. For the workload used in this experiment, Max allows less than two queries to be admitted at the same time (see Fig. 5) since each query requires an average of 1,321 buffers ($F \times$ 1,200 pages for **R** plus one I/O buffer). The tight MPL limit imposed by Max prevents the RTDBS from exploiting its disk and CPU resources to cope with the heavier load as the arrival rate increases from 0.04 to 0.08 queries/sec, which explains why, unlike the other two algorithms, Max's disk utilization barely rises. This ineffective resource usage leads to the observed sharp growth in the miss ratio of Max.

In contrast to Max, MinMax attempts to reduce query miss ratios by increasing the system's MPL. This is achieved at the expense of running queries with memory allocations that are less than their maximum, which increases the demands on the CPU and the disks. By giving queries their minimum required memory, MinMax could admit up to an average of 69 queries at the same time (on the average, the minimum memory requirement per query is $\sqrt{F\|R\|}$ pages + 1 I/O buffer = 37 pages), thus allowing much higher average MPLs as Fig. 5 shows. Moreover, the increased CPU and disk demands that result have little harmful effect here, as the disk utilization barely exceeds 45% even at an arrival rate of 0.08 queries/sec, indicating that there are abundant CPU and disk capacities to service all the admitted queries. The overall result is that MinMax uses the system's resources in a much more effective fashion than Max. Consequently, the higher execution times that MinMax produces are more than compensated for by the large reduction in admission waiting times, thus resulting in total response times that are significantly lower than the response times of Max. This accounts for MinMax's superior miss ratios in Fig. 3.

We now turn our attention to the PAQRS algorithm. In order to understand how PAQRS adapts itself to the workload, we examine Fig. 6, which traces the target MPL settings of PAQRS over the initial 10 hours of operation at an arrival rate of 0.075 queries/sec. PAQRS starts with Max, but it quickly detects that this allocation strategy is not satisfactory because it leads to a very limited MPL while leaving the CPU and disks grossly underutilized. This causes PAQRS to switch to MinMax mode to make a higher MPL possible. The target MPL is first set to 25, following the suggestion of the Resource Utilization heuristic. Once PAQRS has gathered three miss ratio observations, it invokes the miss ratio projection method, which quickly steers the target MPL to the vicinity of 10 where it stabilizes. This MPL is sufficiently loose to admit all of the queries into the system most of the time. Indeed, Fig. 5 shows that PAQRS consistently achieves high MPL settings, thus enabling it to behave like the MinMax algorithm. This is why PAQRS manages to closely match the performance of MinMax, which offers the best miss ratios for this experiment.

Besides this experiment, we have also carried out experiments where we varied the workload and resource parameter settings [22]. In particular, we studied the performance of PMM (i.e., single-class PAQRS) under higher disk contention levels, workload fluctuations, and workloads that contained external sorts. In all of these experiments, the algorithm was able to consistently deliver low miss ratios by dynamically reaching the right compromise between Max and MinMax, and by setting an appropriate target MPL.

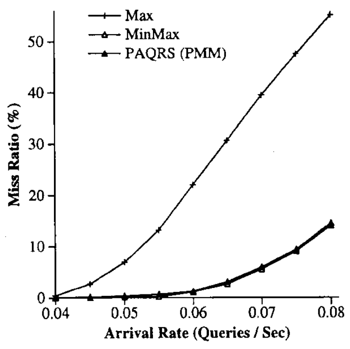


Fig. 3. Miss ratio (single-class).

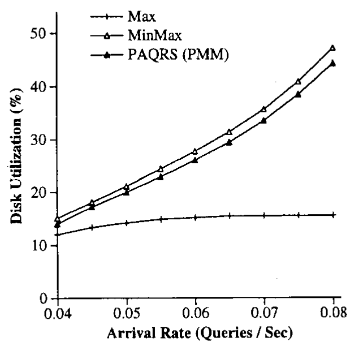


Fig. 4. Disk utilization (single-class).

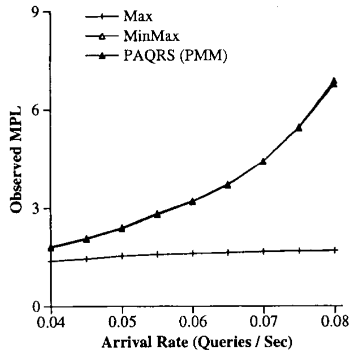


Fig. 5. MPL (single-class).

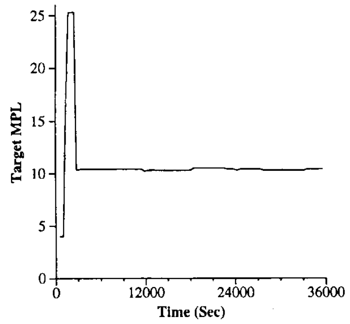


Fig. 6. PAQRS MPL, λ = 0.075 (single-class).

B. Baseline Multiclass Workload Experiment

Having gained some initial intuition regarding the behavior of the MPL control and memory allocation mechanisms of PAQRS, we now proceed to evaluate its ability to handle multiclass workloads. Our baseline experiment uses a workload that consists of two classes of hash joins, Small and Medium. With the exception of the arrival rate, which is fixed at 0.065 queries/sec, the characteristics of the Medium class are the same as those of the previous experiment. For the Small class, $\|R\|$ ranges between 50 and 150 pages, while $\|S\|$ ranges from 250 to 750 pages. The slack ratio interval for Small joins is also set to [2.5, 7.5], and the arrival rate of this class, λ_{Small}, ranges from 0 to 1.2 queries/sec. Table VIII summarizes the detailed database and workload characteristics. The performance objective for the baseline experiment is to **balance** the miss ratio of the two classes, i.e., *RelMissRatio* = {1:1}. The number of disks is raised to 12 to accommodate the heavier load here, while the rest of the resource parameters remain at their settings from the previous experiment.

TABLE VIII
DATABASE AND WORKLOAD PARAMETER SETTINGS FOR THE BASELINE EXPERIMENT

Database	Meaning	Setting
NumGroups	Number of relation groups in the database	4
$RelPerDisk_1$	Number of relations per disk for group 1	3
$SizeRange_1$	Range of relation sizes for group 1	[600, 1800] pages
$RelPerDisk_2$	Number of relations per disk for group 2	3
$SizeRange_2$	Range of relation sizes for group 2	[3000, 9000] pages
RelPerDisk3	Number of relations per disk for group 3	3
$SizeRange_3$	Range of relation sizes for group 3	[50, 150] pages
$RelPerDisk_4$	Number of relations per disk for group 4	3
$SizeRange_4$	Range of relation sizes for group 4	[250, 750] pages
TupleSize	Tuple size of relations in bytes	256 bytes

Workload	Meaning	Setting
NumClasses	Number of classes in the workload	2
$QueryType_1$	Type of class 1 queries	Hash join
$RelGroup_1$	Operand relation groups for class 1 queries	{1, 2}
λ_1	Arrival rate of class 1 queries	0.065
$SRInterval_1$	Range of slack ratios for class 1 queries	[2.5, 7.5]
$QueryType_2$	Type of class 2 queries	Hash join
$RelGroup_2$	Operand relation groups for class 2 queries	{3, 4}
λ_2	Arrival rate of class 2 queries	varied from 0 to 1.2
$SRInterval_2$	Range of slack ratios for class 2 queries	[2.5, 7.5]
F	Fudge factor for hash joins	1.1

Figs. 7 and 8 plot the class miss ratios and system miss ratios produced by PMM and PAQRS as a function of the arrival rate of the Small class. In order to understand the behavior of the various mechanisms of PAQRS, we also include in the figures curves that are labeled *PAQRS(NoBiasCtrl)*, which shows how PAQRS would perform without its bias control mechanism. The figures show that while PMM clearly delivers the lowest system miss ratios, it is also extremely biased, penalizing the Medium class as the load from the Small class increases: as λ_{Small} increases from 0 to 1.2 queries/sec, the miss ratio of the Small class barely rises, but the miss ratio of the Medium class increases dramatically, growing from a low of near-zero misses to a high of 70%. In comparison, PAQRS without bias control and the full PAQRS algorithm come much closer to achieving balanced miss ratios, though at the expense of higher system miss ratios. In fact, the full PAQRS algorithm exhibits virtually no skewed behavior at all. These results clearly demonstrate that the choice of a query scheduling algorithm can have a very significant impact on class miss ratios. To understand the behavior of the two algorithms, we shall analyze each algorithm in turn with the aid of Figs. 9 to 14, which give the weighted miss ratios, observed MPLs, disk utilizations, waiting time ratios (the ratio of the waiting time to the time constraint) and response time ratios (the ratio of the total response time to the time constraint) for both the Small and Medium classes, and the percentage of queries in each class that are assigned to the PAQRS reserve group. In computing the average response time ratios, a late query is considered to have a response time ratio of 100% since the query is aborted only after its deadline expires. We shall henceforth refer to waiting time ratios and response time ratios collectively as timing ratios.

Let us first examine the PMM algorithm, which treats queries as if they all belonged to a single class. There are two reasons why this leads to a biased treatment of classes. The first reason is that the Earliest Deadline policy used for resource scheduling is inherently biased [19]. When treated on par with the Small queries, Medium queries are assigned lower priorities by ED most of the time because their deadlines are much further in the future. Consequently, Medium queries are not able to compete for resources early in their lifetimes; many of them only gain enough priority after their deadlines become infeasible, thus wasting the resources that they consume. Figs. 12 and 13 provide evidence of this bias in the ED policy. Even at a low load of λ_{Small} = 0.2 queries/sec, Medium queries spend more than 10% of their deadlines waiting for admission and another 35% of their time constraints executing in the system (see Fig. 13). In contrast, Small queries have negligible admission waiting times and finish way ahead of their deadlines (Fig. 12). As the load mounts, the response times of Medium queries rapidly approach their deadlines, while the response times of Small queries rise much more slowly. For example, at λ_{Small} = 1.2 queries/sec, where about 70% of the Medium queries miss their deadlines, the average Small query still manages to complete before even 30% of its time constraint has elapsed. As a result, Small queries fare much better than their Medium counterparts.

Another reason for PMM's biased behavior is that the Small class, by virtue of its higher arrival rate, exerts a disproportional influence on the various measurements that PMM relies upon when making its MPL and memory allocation choices, thus resulting in choices that favor Small queries. Since a Small join query requires an average of only 111 memory pages ($F\|R\|$ pages + 1 I/O buffer = 111 pages) to satisfy its maximum demand, memory contention becomes an issue for the Small class only when the number of queries in the system

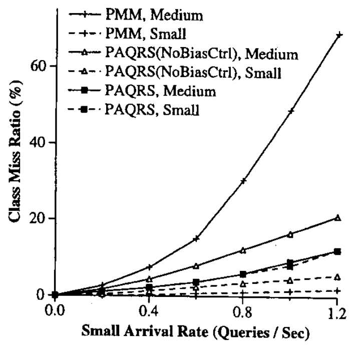


Fig. 7. Class miss ratio (baseline).

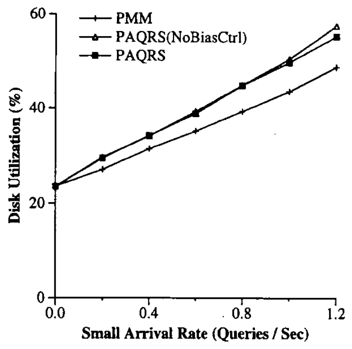


Fig. 11. Disk utilization (baseline).

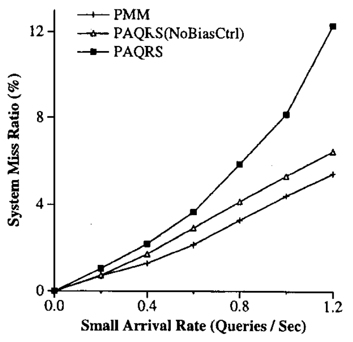


Fig. 8. System miss ratio (baseline).

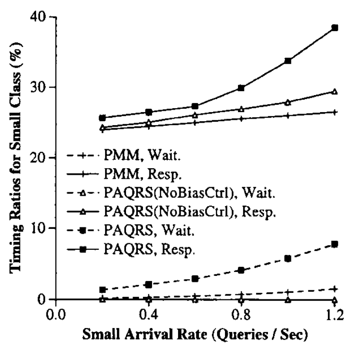


Fig. 12. Small timing ratios (baseline).

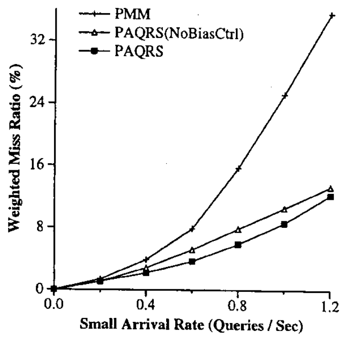


Fig. 9. Weighted miss ratio (baseline).

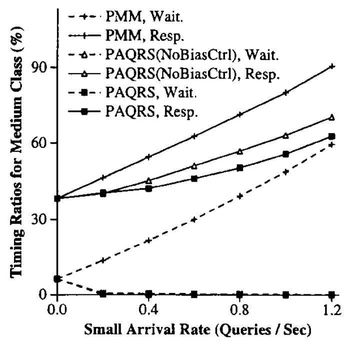


Fig. 13. Medium timing ratios (baseline).

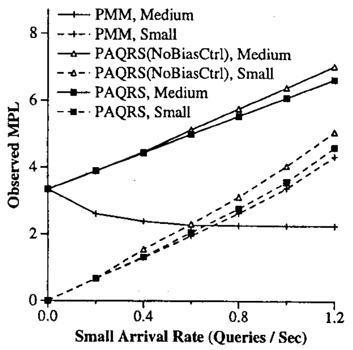


Fig. 10. Observed MPL (baseline).

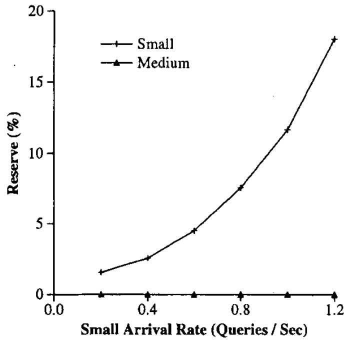


Fig. 14. PAQRS reserve group (baseline).

exceeds 23 at a time (2,560 memory pages divide by 111 pages per query). However, as the low observed MPLs in Fig. 10 show, this is unlikely to happen. PMM therefore concludes that memory contention is negligible and that Max is the preferred memory allocation strategy. This severely limits the MPL of the Medium class. In fact, on the average only two Medium queries get to execute concurrently, as each Medium join query's expected maximum memory requirement is 1,321 pages. Consequently, Medium queries suffer long admission waiting times that cause many of them to miss their deadlines, despite the disks' having excess capacity as the lower PMM disk utilizations in Fig. 11 suggest. In contrast, the Small class benefits tremendously from the choice of the Max strategy. This is because the low concurrency of the Medium class leaves the Small queries with ample memory and virtually all of the CPU and disk capacity that they require. Therefore, Small queries are able to enjoy relatively short admission waiting and response times at the expense of the Medium class under PMM. This bias in MPL and memory allocation strategy choices, together with ED's inherent bias, accounts for the disparity in miss ratios between the two classes.

Having understood the forces that cause PMM to be biased, we now investigate the extent to which PAQRS is able to make MPL and memory allocation strategy choices that are more conducive to achieving balanced class miss ratios, which is the workload objective for this experiment. The higher observed MPLs for both Small and Medium queries produced by PAQRS(NoBiasCtrl) in Fig. 10 show that PAQRS decides to admit more queries and does not insist on maximum allocations here. This virtually eliminates admission waiting time for the Medium class, allowing its queries to enjoy CPU and disk services early in their lifetimes. The heavier disk utilizations in Fig. 11 suggest that the disks are utilized more productively now. As a result, Medium queries are able to complete so much earlier that their miss ratios plummet from PMM's high of nearly 70% at λ_{Small} = 1.2 queries/sec to just over 20% for PAQRS without bias control. However, the improved performance of the Medium class is achieved at the expense of somewhat higher miss ratios for Small queries, whose response times are prolonged by the heightened resource contention. This loss suffered by the Small class to the benefit of the Medium queries is the reason that PAQRS without bias control delivers a more balanced miss ratio distribution and a much lower weighted miss ratio than PMM does. The higher system miss ratio that PAQRS without bias control produces can be explained as follows: Since a Medium query consumes significantly more resources than a Small query, the system is likely to have to sacrifice several Small queries in order to help a Medium query meet its deadline, especially when the load is heavy. This naturally results in higher system miss ratios because every late query, regardless of its class, contributes equally to the system miss ratio. Note that PAQRS would have been discouraged from helping the Medium class had it not adopted the weighted miss ratio to measure overall system efficiency. Instead, being driven by the lower weighted miss ratio measurements that result, PAQRS is able to arrive at the right MPL and memory allocation strategy.

Finally, we turn our attention to the bias control mechanism of the PAQRS algorithm. Fig. 14 shows that this mechanism relegates more and more of the Small queries to the reserve group as λ_{Small} increases. This raises the average admission waiting time of the Small class and leads to a decline in its MPL, as reserve queries are granted admission only after the regular queries from all classes have received their maximum required memory. The higher fraction of reserve queries also lowers the average priority of the Small class, which in turn lengthens its average response time (over and above the delay it already suffers from the Medium class' higher concurrency under the MPL and memory allocation mechanisms of PAQRS) and pushes up its miss ratio. However, as a result of the Small class' lower average priority, Medium queries can now run with more memory. This reduces the amount of temporary (hash bucket) data that Medium queries must write out, which explains why the disk utilizations of the full PAQRS algorithm are lower than those of PAQRS(NoBiasCtrl) in Fig. 11. This also helps Medium queries to complete earlier, bringing their miss ratios down further to match those of the Small class. For example, at λ_{Small} = 1.2 queries/sec, a Medium query requires an average of just over 60% of its time constraint to run when the full PAQRS algorithm is employed, whereas it takes more than 70% of its deadline under PAQRS without bias control. Consequently, the full PAQRS algorithm is able to completely balance the class miss ratios. Interestingly, despite producing lower miss ratios for the Medium class, the full PAQRS algorithm does not improve significantly upon the weighted miss ratios of PAQRS(NoBiasCtrl). This is because PAQRS without bias control already allows the system resources to be utilized productively, so the full PAQRS algorithm has to achieve further reductions in the number of late Medium queries by sacrificing (many more) Small queries rather than by improving the efficiency of resource usage.

To summarize the results of this experiment, we can draw the following conclusions: First, while PMM is very effective in minimizing the system miss ratio, it is also biased in its treatment of different classes. This will be unacceptable for those applications that require controlled miss ratios. Second, by setting the target MPL and memory allocation strategy according to administratively defined workload objectives, PAQRS can come considerably closer to achieving balanced class miss ratios than PMM. Finally, by also manipulating the individual class quotas for regular queries, PAQRS is able to influence their relative miss ratios enough to produce equitable miss ratios.

C. Skewed Class Objectives

Having demonstrated in the previous experiment that PAQRS can successfully achieve balanced class miss ratios, we now explore its ability to meet skewed workload objectives. This is accomplished by varying the algorithm parameter *RelMissRatio*. We first set it to favor the Small class; we then reverse the setting so that Medium queries become more valuable. All of the database and workload parameters remain as they were in the baseline experiment.

For the first part of the experiment, we set *RelMissRatio* to {2:1}, so the target miss ratio distribution is of the form $MissRatio_{Medium}$ = 2x% and $MissRatio_{Small}$ = x%. Figs. 15 and 16 present the resulting class miss ratios and weighted miss ratios, while Fig. 17 plots the ratio of $MissRatio_{Small}$ to $MissRatio_{Medium}$ as a function of λ_{Small}. The figures show that the behavior of both PMM and PAQRS(NoBiasCtrl) are virtually the same as those observed in the baseline experiment. In the case of PMM, this is to be expected, as PMM is not designed to discern class distinctions or to meet multiclass objectives; changes in the *RelMissRatio* parameter naturally have no effect on PMM's behavior. In the case of PAQRS without bias control, its behavior remains essentially unchanged because, even for the {2:1} target miss ratio distribution, it still misses more Medium queries than desired. Consequently, PAQRS without bias control is already using the MPL setting and the memory allocation strategy that are most favorable to the Medium class, as it was in the previous experiment. Not surprisingly, the full PAQRS algorithm successfully achieves the {2:1} target distribution; in fact, it is an easier target than the objective of balanced miss ratios in the previous experiment since it requires a smaller improvement in the miss ratio of the Medium class.

For the second part of the experiment, we reverse the target miss ratio distribution to the more challenging setting of *RelMissRatio* = {1:2}. The resulting class miss ratios, weighted miss ratios, and $MissRatio_{Small}$ to $MissRatio_{Medium}$ ratios are presented in Figs. 18 to 20. These figures show that while selecting the appropriate MPL and memory allocation settings almost enabled PAQRS without bias control to meet the target miss ratio distribution of *RelMissRatio* = {2:1} earlier, PAQRS is not able to improve the relative miss ratio of the Medium class any further without its bias control mechanism. Without this mechanism, PAQRS fails miserably here, producing $MissRatio_{Small}/MissRatio_{Medium}$ values that are far short of the target. In contrast, the full PAQRS algorithm again attains the target distribution. Even at high λ_{Small} values, where the workload consists predominantly of Small queries, and where Medium queries are in a very disadvantaged position due to heavy contention from Small queries that have nearer deadlines, the full PAQRS algorithm still manages to bring the miss ratios of the Medium class down to meet the demanding workload objective. However, the full PAQRS algorithm produces only slightly lower weighted miss ratios than PAQRS(NoBiasCtrl) here (Fig. 19). As discussed in the baseline experiment, this is because the resource consumption of Medium queries is much more than that of the Small queries, so the system has to sacrifice many more Small queries to reduce the number of late Medium queries.

To summarize, this experiment confirms that PMM is incapable of achieving the target miss ratio distribution of multiclass workloads. In contrast, the MPL, memory allocation, and bias control mechanisms of PAQRS are able to work in unison to consistently meet multiclass performance objectives, whether balanced or skewed.

D. Identical Classes

In the first two experiments, we saw that the bias control mechanism of PAQRS is very effective in regulating per-class performance to achieve a desired target miss ratio distribution, even despite the classes' very different characteristics. However, this mechanism could impose a cost, as it may be overly conservative in setting its regular query quotas; this would cause too many queries to be assigned to the reserve group, resulting in unnecessary deadline misses. To explore this potential drawback, we now replace the Small class in the baseline experiment with another class that is identical to the Medium class, and we equate the mean arrival rates of the two classes. The rest of the parameters are set as in the baseline experiment. Finally, *RelMissRatio* is set to {1:1} i.e., the target is to balance the miss ratios of the classes.

Fig. 21 plots the system miss ratios produced by PMM and PAQRS as a function of λ_{Small}. This figure shows that PAQRS produces slightly higher system miss ratios than PMM, indicating that the bias control mechanism of PAQRS indeed becomes a slight liability here. This occurs because the two classes only experience similar *average* miss ratios. At any particular instant, workload fluctuations will inevitably cause the two class miss ratios to deviate from each other; in reaction to these deviations, PAQRS will relegate some queries from the class that appears to be overachieving to the reserve group. While only a small percentage of the queries are affected, there is nonetheless some overhead involved. Fortunately, PAQRS suffers only a slight performance deterioration as a result. For example, at arrival rates of 0.06 queries/sec, where both classes are missing as many as 29% of their queries under PMM, PAQRS misses just about 30% of the queries. Consequently, while PAQRS can lead to some small overhead, its benefit of achieving the target miss ratio distribution more than justifies its use.

E. Workload Changes

The preceding experiments lead us to the conclusion that PAQRS is very effective for relatively stable real-time workloads. The objective of this next experiment is to find out how well PAQRS reacts to dynamic workload *changes*. This is done by subjecting the various query scheduling algorithms to a workload whose composition changes every X simulated hours, where X varies randomly (and uniformly) between two and five. At any given time, the workload contains two of the following three query classes—Small, Medium, and Sort. The Small and Medium classes are the same as in the baseline experiment. Each query in the Sort class sorts a single relation **R**, where $\|R\|$ ranges from 600 to 1,800 pages. Table IX summarizes the database and workload parameters (except arrival rates). The class arrival rates vary from one workload mix to another. To highlight the performance tradeoffs between the algorithms, they are chosen so that the average miss ratios produced by the best algorithm(s) in each case are in the neighborhood of 5 to 10%. The chosen arrival rates are listed in Table X, while the resource parameters are the same as in

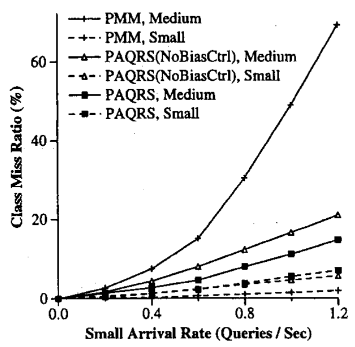


Fig. 15. Class miss ratio (1:2).

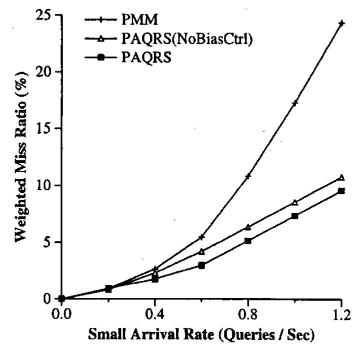


Fig. 16. Weighted miss ratio (1:2).

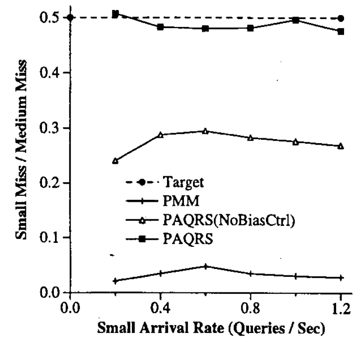


Fig. 17. Class miss ratio dist. (1:2).

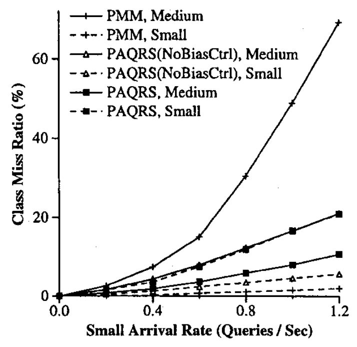


Fig. 18. Class miss ratio (2:1).

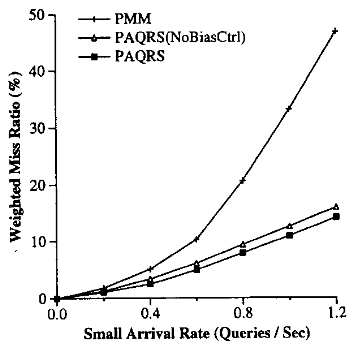


Fig. 19. Weighted miss ratio (2:1).

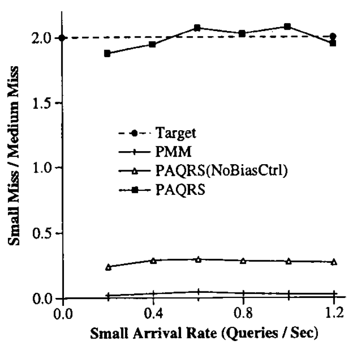


Fig. 20. Class miss ratio dist. (2:1).

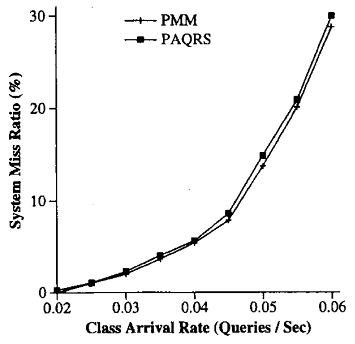


Fig. 21. System miss ratio (identical).

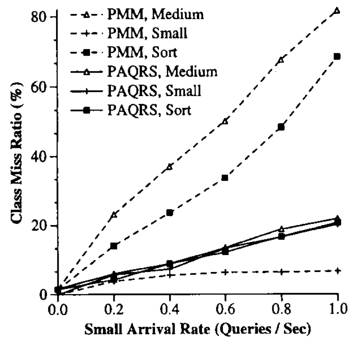


Fig. 22. Class miss ratio (three classes).

the baseline experiment. To ensure that all of the workload mixes are tried in a relatively short simulated time period of 45 hours, the workload repeatedly cycles through the three possible mixtures, i.e., it starts with mixture #1, goes on to mixture #2, which is followed by mixture #3, then returns to mixture #1, and so on. Our target is to balance the miss ratios of the two classes within each workload mix.

TABLE IX
DATABASE AND WORKLOAD PARAMETER SETTINGS (WORKLOAD CHANGES)

Database	Value	Workload	Value
NumGroups	4	*QueryType*$_{Medium}$	Hash join
RelPerDisk$_1$	3	*RelGroup*$_{Medium}$	{1, 2}
SizeRange$_1$	[600, 1800] pp.	*SRInterval*$_{Medium}$	[2.5, 7.5]
RelPerDisk$_2$	3	*QueryType*$_{Small}$	Hash join
SizeRange$_2$	[3000,9000]pp.	*RelGroup*$_{Small}$	{3, 4}
RelPerDisk$_3$	3	*SRInterval*$_{Small}$	[2.5, 7.5]
SizeRange$_3$	[50, 150] pp.	*QueryType*$_{Sort}$	External sort
RelPerDisk$_4$	3	*RelGroup*$_{Sort}$	{1}
SizeRange$_4$	[250, 750] pp.	*SRInterval*$_{Sort}$	[2.5, 7.5]

TABLE X
CLASS ARRIVAL RATES IN QUERIES/sec (WORKLOAD CHANGES)

Workload Mix	Small	Medium	Sort
1	1.0	0.065	—
2	1.0	—	0.08
3	—	0.045	0.06

Table XI summarizes the performance of the three classes in the form of average class miss ratios. We shall examine these results according to workload mixes. Although workload mixture #1 has exactly the same composition as the workload used in the baseline experiment, both PMM and PAQRS produce higher miss ratios here than they did previously. This is due to the introduction of workload changes, which cause each of the algorithms to reset themselves. Consequently, the algorithms need to adapt to the workload repeatedly, and inefficient resource usage during the adjustment periods pushes up the miss ratios. Other than the higher miss ratios, the qualitative tradeoffs between the two algorithms remain the same. In particular, PAQRS still achieves the target miss ratio distribution.

Turning our attention to workload mixture #2, we first note that PMM again discriminates against the queries that have larger memory demands. In fact, the Sort queries in this workload mix perform significantly worse than the Medium queries in workload mixture #1. This is because while the memory demands of the Sort queries and Medium queries are about the same, the load that the Sort queries place on the disks and the CPU is considerably lighter; on the average, each Sort query only has to sort one 120-page relation, whereas the average Medium query has to join a 120-page relation with a 600-page relation. Consequently, memory is a much more critical resource for workload mixture #2, thus amplifying the biased behavior of the Max strategy that PMM chooses. In contrast, PAQRS again manages to balance the class miss ratios.

Finally, for workload mixture #3, PMM adopts the MinMax mode and high MPL settings to service the two memory-intensive classes. Its slightly skewed miss ratios are a result of ED favoring the Sort queries, which are somewhat shorter than the Medium hash join queries. This biased behavior is rectified by the bias control mechanism of PAQRS. This experiment shows that PAQRS not only performs well under stable workloads, but is also capable of adapting to workload changes.

TABLE XI
AVERAGE CLASS MISS RATIOS (WORKLOAD CHANGES)

	PMM			PAQRS		
Mixture	Small	Medium	Sort	Small	Medium	Sort
1	1.6%	44.3%	-	9.1%	9.6%	-
2	1.3%	-	79.0%	7.4%	-	7.3%
3	-	10.1%	11.4%	-	10.9%	10.8%

F. A Three-Class Workload

Up to this point, we have examined the performance of PAQRS using workloads that consisted of only two classes in order to simplify our discussions. However, PAQRS is intended to be a general multiclass query scheduling algorithm, and is not limited to handling only simple workloads. To demonstrate that PAQRS is capable of managing more complex workloads well, we conclude this section by repeating the baseline experiment using a workload that is made up of three different classes. We use the same three classes that we used in the previous experiment; instead of choosing only two out of three classes at a time, however, we activate all three classes *concurrently*. The arrival rate of the Sort and Medium classes are both set to 0.045 queries/sec, while the arrival rate of the Small class is varied.

The class miss ratios of the three query scheduling algorithms for this workload are shown in Fig. 22. The performance trends in these figures reveal no surprises: PMM still affords the Small class favored treatment at the expense of the two memory-intensive classes. Among these two classes, the more resource-demanding Medium class suffers a higher miss ratio because of the inherent bias of the Earliest Deadline scheduling policy. Again, we see that PAQRS is able to manipulate the priority of the classes appropriately to achieve the target miss ratio distribution.

G. Scalability of Results

In order to limit simulation costs, we intentionally chose to use small relation and memory sizes in our experiments. This raises questions about the scalability of our results to larger systems: How would larger memory and relation sizes affect the performance of the various algorithms? Would PAQRS still be able to perform as well as it did? To verify the scalability of our results, we carried out two different sets of experiments—a set of medium-scale experiments, reported in this paper, and a set of small-scale experiments that involved database and memory sizes that were 10 times smaller. The two sets of experiments produced essentially the same qualitative algorithm behavior; in other words, our results scaled up from small database and memory sizes to medium sizes. We therefore expect our results to scale up to even larger memory and relation sizes; PAQRS should be just as effective for larger systems as it was for the workloads and configurations that we have experimented with here.

VII. Conclusion

In this paper, we have continued and extended our previous study on the problem of scheduling queries in firm real-time database systems (RTDBS), which we reported in [22]. In that study, we proposed a *Priority Memory Management* (PMM) algorithm that aims to minimize the number of missed deadlines by adapting both the multiprogramming level (MPL) and the memory allocation strategy of an RTDBS according to feedback on system behavior. This eliminates the need for any advance knowledge of workload characteristics or query execution times. Instead, the setting of the MPL is determined primarily by a statistical projection method, called miss ratio projection, which is supplemented by a resource utilization heuristic when the statistical method fails. PMM incorporates two memory allocation strategies—a Max strategy under which each query receives either its maximum required memory or no memory at all, and a MinMax strategy that allows some queries to run with their minimum required memory while others get their maximum. Both strategies employ the Earliest Deadline (ED) policy so that queries whose deadlines are more imminent are given memory ahead of queries that are less urgent. The choice of memory allocation strategy is based on statistics about the workload characteristics that PMM gathers. In order to ensure that its MPL setting and memory allocation strategy choices remain appropriate, PMM constantly monitors the workload for changes that may necessitate adjustments to those decisions. Experimental results obtained with a detailed RTDBS simulation model, which appeared in [22] and which we summarized briefly here, indicate that the admission control and memory allocation mechanisms of PMM are very effective in helping an RTDBS achieve low deadline misses. However, when presented with a multiclass workload, PMM can produce skewed class miss ratios that may be unacceptable for some applications.

In order to better meet multiclass performance objectives, as expressed in the form of target miss ratio distributions, this paper has extended PMM to create a new algorithm called *Priority Adaptation Query Resource Scheduling* (PAQRS). PAQRS modifies the MPL and memory allocation strategy selection mechanisms of PMM to pick a global MPL setting and a system-wide memory allocation strategy that are conducive to achieving the given target distribution. It then regulates the MPL and memory allocation of individual classes indirectly by controlling the priority of their queries. This regulation is accomplished by dividing the queries in an RTDBS into two priority groups—a regular group and a reserve group—and by setting a quota of regular queries for each class. All regular queries are assigned higher priorities than any reserve query, so PAQRS manipulates the relative priority of individual classes simply by adjusting their regular query quotas. By appropriately setting these quotas, PAQRS is able to influence the miss ratios of the classes to conform to the target distribution.

Through a series of simulation experiments, we demonstrated that the modified MPL and memory allocation strategy selection mechanisms of PAQRS enable it to utilize the system resources efficiently to reduce the overall number of deadline misses. However, these mechanisms alone are inadequate for regulating the distribution of deadline misses among multiple query classes. This inadequacy is overcome by the PAQRS algorithm's bias control mechanism. Hence, all three mechanisms are important in helping PAQRS achieve its given performance objective. Finally, PAQRS was shown to be able to adapt to the offered workload quickly enough so that it can work well even when workload changes sometimes occur; of course, were the workload to fluctuate too rapidly, PAQRS' performance would likely deteriorate with increased workload fluctuations. While we only experimented with queries that perform either external sorting or hash join operations, PAQRS is designed to schedule general query workloads effectively by balancing their demands on the system's memory, CPU, and disks. In particular, PAQRS can be extended to handle complex database queries that use external sorting and hash joins as building blocks, such as queries with aggregates, group-by clauses, and/or order-by clauses. Therefore, we conclude that PAQRS should be very useful for scheduling complex query workloads in an RTDBS.

A number of open issues remain in the area of real-time query scheduling. We have considered only workloads involving mixes of queries in this paper; RTDBS workloads are likely to contain transactions as well as queries. Thus, it would be useful to combine PAQRS with long-term data buffering techniques, such as those proposed in [5], in order to provide a truly complete memory manager for RTDBSs. The concurrent execution of long-running queries and short transactions also raises concurrency control issues that need to be resolved. Another avenue for future work is to explore ways to shorten the adjustment time of PAQRS by incorporating more sophisticated MPL control and memory allocation heuristics. This would help to improve PAQRS' ability to adapt to workload changes. Finally, we would like to apply the techniques that we developed for PAQRS to nonreal-time environments such as the goal-oriented database system environment studied in [6].

Acknowledgment

This work was carried out as part of HweeHwa Pang's PhD research at the University of Wisconsin at Madison.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," *Proc. 14th Int'l Conf. Very Large Data Bases*, Aug. 1988.

[2] R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions with disk resident data," *Proc. 15th Int'l Conf. Very Large Data Bases*, Aug. 1989.

[3] R. Abbott and H. Garcia-Molina, "Scheduling I/O requests with deadlines: A performance evaluation," *Proc. 11th IEEE Real-Time Systems Symp. (RTSS)*, Dec. 1990.

[4] D. Bitton and J. Gray, "Disk Shadowing," *Proc. 14th Int'l Conf. Very Large Data Bases*, Aug. 1989.

[5] K.P. Brown, M.J. Carey, and M. Livny, "Managing memory to meet multiclass workload response time goals," *Proc. 19th Int'l. Conf. Very Large Data Bases*, Aug. 1993.

[6] K.P. Brown, M. Mehta, M.J. Carey, and M. Livny, "Towards automated performance tuning for complex workloads," *Proc. 20th Int'l Conf. Very Large Data Bases*, Sept. 1994.

[7] S. Chen, J.A. Stankovic, J.F. Kurose, and D. Towsley, "Performance evaluation of two new disk scheduling algorithms for real-time systems," *J. Real-Time Systems*, vol. 3, no. 3, Sept. 1991.

[8] D. Cornell and P. Yu, "Integration of buffer management and query optimization in a relational database environment," *Proc. 15th Int'l Conf. Very Large Data Bases*, Aug. 1989.

[9] D.L. Davison and G. Graefe, "Memory-contention responsive hash joins," *Proc. 20th Int'l Conf. Very Large Data Bases*, Sept. 1994.

[10] J.L. Devore, *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole Publishing Co., pp. 283-301, 326-335, 1991.

[11] N.R. Draper and H. Smith, *Applied Regression Analysis*. John Wiley & Sons, pp. 70-136, 1981,.

[12] J.R. Haritsa, M.J. Carey, and M. Livny, "On being optimistic about real-time constraints," *Proc. 1990 ACM PODS Symp.*, Apr. 1990.

[13] J.R. Haritsa, M. Livny, and M.J. Carey, "Earliest deadline scheduling for real-time database systems," *Proc. 12th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 1991.

[14] J. Huang, J.A. Stankovic, D. Towsley, and K. Ramamritham, "Experimental evaluation of real-time transaction processing," *Proc. 10th IEEE Real-Time Systems Symp. (RTSS)*, Dec. 1989.

[15] W. Kim and J. Srivastava, "Enhancing real-time DBMS performance with multiversion data and priority based disk scheduling," *Proc. 12th IEEE Real-Time Systems Symp. (RTSS)*, Dec. 1991.

[16] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, Jan. 1973.

[17] M. Livny, "DeNet User's Guide, Version 1.5," Computer Sciences Dept., Univ. of Wisconsin, Madison, 1990.

[18] M. Nakayama, M. Kitsuregawa, and M. Takagi, "Hash-partitioned join method using dynamic destaging strategy," *Proc. 14th Int'l Conf. Very Large Data Bases*, Aug. 1988.

[19] H. Pang, M. Livny, and M.J. Carey, "Transaction scheduling in multiclass real-time database systems," *Proc. 13th IEEE Real-Time Systems Symp. (RTSS)*, Dec. 1992.

[20] H. Pang, M.J. Carey, and M. Livny, "Partially preemptible hash joins," *Proc. ACM SIGMOD Conf.*, May 1993.

[21] H. Pang, M.J. Carey, and M. Livny, "Memory-adaptive external sorting," *Proc. 19th Int'l Conf. Very Large Data Bases*, Aug. 1993.

[22] H. Pang, M.J. Carey, and M. Livny, "Managing memory for real-time queries," *Proc. ACM SIGMOD Conf.*, May 1994.

[23] K. Ramamritham, "Real-time databases," *Distributed and Parallel Databases*, vol. 1, no. 2, Apr. 1993.

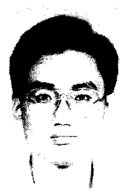
[24] R. Sargent, "Statistical analysis of simulation output data," *Proc. Fourth Ann. Symp. Simulation Computer Systems*, Aug. 1976.

[25] L.D. Shapiro, "Join processing in database systems with large main memories," *ACM Trans. Database Systems*, vol. 11, no. 3, Sept. 1986.

[26] J.A. Stankovic and W. Zhao, "On real-time transactions," *ACM SIGMOD Record*, vol. 17, no. 1, Mar. 1988.

[27] P.S. Yu and D.W. Cornell, "Buffer management based on return on consumption in a multiquery environment," *VLDB J.*, vol. 2, no. 1, Jan. 1993.

[28] H. Zeller and J. Gray, "An adaptive hash join algorithm for multiuser environments," *Proc. 16th Int'l Conf. Very Large Data Bases*, Aug. 1990.



HweeHwa Pang received the BS–with first class honors–and MS degrees from the National University of Singapore in 1989 and 1991, respectively, and the PhD degree from the University of Wisconsin at Madison in 1994, all in computer science. His PhD research focus was on adaptive query processing and resource scheduling in database management systems, in particular real-time database systems. He is currently a member of the associate research staff at the Institute of Systems Science, National University of Singapore, where he is building a multimedia storage server. His research interests include database management systems, multimedia systems, and real-time systems.

Michael J. Carey received the BS degree in electrical engineering and mathematics and the MS degree in electrical engineering (computer engineering) from Carnegie Mellon University in 1979 and 1981, respectively. He received the PhD degree in computer science from the University of California at Berkeley in 1983. He then became a faculty member in the Computer Science Department of the University of Wisconsin at Madison. Dr. Carey spent the summer of 1989 and the 1993–1994 academic year as a visiting scientist at the IBM Almaden Research Center, where he is now on the staff.

Dr. Carey's research interests include object-oriented database systems, parallel and distributed databases, database systems performance, and database applications that involve user-specified performance objectives. He was a co-principal investigator of the EXODUS extensible DBMS project. He is now involved in SHORE, a successor to the EXODUS project that has the goal of replacing the Unix file system with a persistent object repository that can work effectively across a wide variety of hardware platforms and application programming languages. During his first visit to IBM, he worked on the Starburst extensible DBMS project. During his second visit, he helped to start the *Garlic* project, a new research effort in the area of multimedia information system.

Dr. Carey received an IBM Faculty Development award in 1984, an Incentives for Excellence award from Digital Equipment Corporation in 1986, and a National Science Foundation Presidential Young Investigator award in 1987. He is a member of the IEEE and the ACM, an associate editor of *ACM Transactions on Database Systems*, and the secretary/treasurer of the ACM SIGMOD group.

Miron Livny received the BS degree in physics and mathematics in 1975 from the Hebrew University, Israel, and the MSc and PhD degrees in computer science from the Weizmann Institute of Science, Israel, in 1978 and 1984, respectively. Since 1983 he has been on the faculty of the Computer Science Department of the University of Wisconsin at Madison, where he is currently a professor.

Dr. Livny's research focuses on scheduling policies for processing and data mangagement systems and on tools that can be used to evaluate such policies. His recent work includes real-time DBMSs, client server systems, batch processing, and tools for experiment management.