Priority in DBMS Resource Scheduling

Michael J. Carey
Rajiv Jauhari
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT — In this paper, we address the problem of priority scheduling in a database management system. We start by investigating the architectural consequences of adding priority to a DBMS. Specific priority-based schemes are then proposed for managing DBMS resources, including a priority-based disk scheduling algorithm and two approaches to adding priority to the buffer manager of a DBMS. We study the performance of our proposals through simulation, both individually and in combination. Our simulation results indicate that the objectives of priority scheduling cannot be met by a single priority-based scheduler. They show that, regardless of whether the system bottleneck is the CPU or the disk, priority scheduling on the critical resource must be complemented by a priority-based buffer management policy.

1. INTRODUCTION

1.1. Background

Priority scheduling of computer systems is a concept that has been studied extensively for more than two decades [Coff68, Klei76]. The priority of a task is a measure of its "importance" to the system, and the objective of priority scheduling is to provide preferential treatment to tasks with higher priority values. One use of priority is as an end in itself. In this case, task priorities are determined externally, and are used directly to guide the system in making scheduling decisions. For example, a system designer may decide to run interactive tasks at a higher level of priority than long batch jobs in order to ensure good responsiveness for interactive users. An alternative use of priority is as a means to an end. Here, task priorities are determined internally (i.e., assigned by the system itself) in order to meet some other goal. For example, priority scheduling can be used as a way of minimizing the number of missed deadlines in a soft real-time computing environment [Chan85].

This work was performed under subcontract to Xerox Advanced Information Technology, which was supported by the Defense Advanced Research Projects Agency and by the Rome Air Development Center under Contract No. F30602-87-C-0029. Additional support was provided by the National Science Foundation under grant IRI-8657323 and by the Digital Equipment Corporation through its Initiatives for Excellence program. The views and conclusions contained in this report are those of the authors and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Rome Air Development Center, or the U.S. Government.

Despite the attention that the concept of priority has received for CPU scheduling in operating systems, it has received very little attention in the database system area. This is somewhat surprising, as priority seems every bit as useful and applicable in a DBMS. The interactive versus batch job example is relevant; in a transaction processing system, it would seem equally desirable to ensure responsiveness for interactive DBMS users while allowing batch jobs to be run as well. In addition, the database and real-time systems communities are beginning to show interest in applying database technology to data-intensive, real-time applications such as stock trading, computer-integrated manufacturing, and command and control systems [Abbo88, SIGM88, Stan88].

Priority scheduling in a DBMS differs from the problem of priority CPU scheduling due to the heterogeneity and multiplicity of DBMS resources. Important DBMS resources include the CPU, disks, and main memory, which are physical resources, and data items, which can be viewed as logical resources. It is common for each of these resources to be managed by an independent scheduler: the underlying operating system kernel schedules the CPU; the operating system or DBMS device drivers handle disk request scheduling; the buffer manager controls the main memory resource (i.e., buffer pool page frames); and, the concurrency control manager schedules accesses to data items. Thus, in a DBMS, there are actually four schedulers into which one could consider incorporating priority.

1.2. Related Work

As mentioned above, priority CPU scheduling for centralized systems has been studied extensively. Overviews of much of this work can be found in [Coff68, Klei76]. Classical approaches to scheduling a single CPU in the presence of priority involve maintaining the CPU queue as a priority queue. Requests are served in priority order either non-preemptively, which is called "head-of-line" or "non-preemptive priority" scheduling, or preemptively, which is known as "preemptive-resume" scheduling. The objective of the latter approach is to ensure that a higher priority job is never made to wait while a lower priority job receives service. Recent work has addressed priority scheduling in the presence of preemption overheads and multiple processors (e.g., in distributed systems [Chan85, Chan87]). Static and dynamic approaches to CPU scheduling for real-time systems have also been studied; this work is reviewed briefly in [Stan88], and [Jens86] provides an excellent treatment of dynamic CPU scheduling approaches for real-time systems. Resources other than the CPU have, for the most part, been ignored with respect to dynamic priority and real-time scheduling [Coff68, Stan88].

In the database area, real-time issues are just beginning to attract attention. [Wede86] proposed a scheme for prefetching data pages for canned transactions in a real-time DBMS; detailed a priori knowledge of the workload is assumed, and static transaction pre-analysis techniques are applied to it. Others have also begun to address real-time DBMS issues [SIGM88, Daya88, Buch89], especially in the area of concurrency control algorithms [Abbo88, SIGM88, Buch89]. Most notably, [Abbo88] proposed and investigated the performance of several priority-based locking (and CPU scheduling) schemes in the context of a memory-resident, real-time DBMS. However, the question of how priority can be used in managing the *physical* resources of a DBMS has not been addressed to the best of our knowledge.

1.3. Our Work

In this paper, we explicitly address the problem of priority scheduling in a DBMS. In particular, we are interested in how priority can be employed in managing the physical resources of a DBMS. The objective of our work is to develop and evaluate policies that provide the best possible service to higher-priority transactions while minimizing the negative impact on lower-priority transactions. Essentially, we would like the DBMS to behave as much like a preemptive-resume server as possible. That is, we would like it to give transactions of a given priority the same service that they would experience in a system that does not serve transactions of lower priority. Thus, the goals of this paper are three-fold. First, we investigate the architectural consequences of adding priority to a DBMS. Second, we develop specific priority-based algorithms for managing the key DBMS resources, especially the disk(s) and the buffer pool. Third, we study the performance of our proposed algorithms, both individually and in combination with one another.

Figure 1.1 depicts the overall architecture of a DBMS from a resource perspective. From the figure, it can be seen that there are several places where a DBMS makes resource scheduling decisions into which priority could be incorporated. First, a DBMS often controls the total number of transactions allowed to be active at any one time, requiring additional transactions to wait outside the system until they can be admitted, in order to prevent thrashing due to buffer pool over-utilization [Chou85]. Second, as discussed earlier, the DBMS or the underlying operating system must make CPU scheduling decisions; similarly, disk

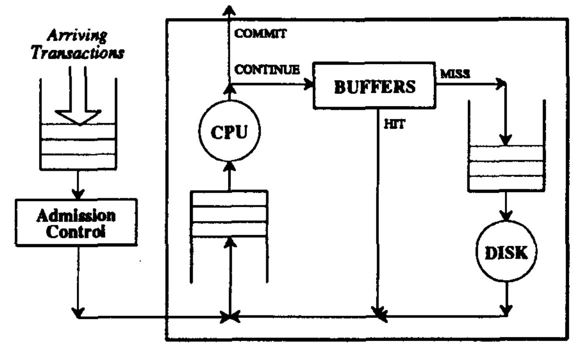

Figure 1.1: DBMS Resource Architecture.

scheduling decisions must be made. Finally, the buffer manager must make page replacement decisions when a non-resident page is requested and there are no free page frames.

In order to see how adding priority at some or all of these decision points might enable us to reduce the response time of high priority transactions, let us consider the composition of a transaction's response time (T_R). Basically, there are two major components of this time, the external waiting time due to admission control (T_{W_EXT}), and the time spent inside the system (T_{SYS}). T_{SYS} itself has two components, CPU time (T_{CPU}) and disk time (T_{DISK}).[1] The CPU time for a transaction consists of two sub-components, the waiting time due to CPU contention (T_{W_CPU}) and the actual CPU service time (T_{S_CPU}). The disk time for a transaction is the product of the number of disk requests that it makes (N_{DISK}) and the average disk access time; the disk access time also consists of a waiting time (T_{W_DISK}) and a service time (T_{S_DISK}). Thus, we see that:

$$T_R = T_{W\_EXT} + T_{W\_CPU} + T_{S\_CPU} + N_{DISK} * (T_{W\_DISK} + T_{S\_DISK})$$

Incorporating priority into the admissions control decision, by ordering waiting transactions according to priority and preempting lower-priority transactions in favor of higher-priority transactions, is a way of reducing T_{W_EXT} for high-priority transactions. In terms of their CPU time, priority-based CPU scheduling can reduce T_{W_CPU} for high-priority transactions; T_{S_CPU} cannot be reduced, however. Similarly, the disk waiting time T_{W_DISK} for high-priority transactions can be reduced through the priority-based scheduling of disk requests; this may actually lead to an increase in T_{S_DISK}, though, as using a non-positional disk scheduling criterion may increase the expected seek time. Finally, buffer replacement decisions can potentially reduce N_{DISK} for high-priority transactions. Of course, none of these reductions for high-priority transactions will come for free. Rather, we must expect corresponding increases in the response time components for transactions of lower priority.

[1] Since our concern is physical resource scheduling, not concurrency control, we focus on read-only queries in this paper. Lock waiting time is therefore not a potential component of response time here.

The remainder of the paper is organized as follows: Section 2 discusses approaches to priority scheduling for the CPU, disk, and buffer resources. This section includes new, priority-based algorithms for disk scheduling and buffer management. Section 3 describes a simulation model for studying the performance consequences of our proposed algorithms. Section 4 presents a series of performance experiments that demonstrate the effectiveness and importance of the various algorithms under workloads of varying intensities for the different resources. Lastly, Section 5 summarizes the contributions of the paper and describes our plans for future work.

2. PRIORITY-BASED RESOURCE MANAGEMENT ALGORITHMS

As described in the introduction, there are three types of physical resources in a DBMS: the processors, the disks, and the main memory buffers. In this section, we describe priority scheduling algorithms for each of these resource types. We present a known algorithm for priority CPU scheduling, and we extend known algorithms for disk scheduling and buffer management to enable them to handle priorities.

2.1. CPU SCHEDULING

A number of options for priority CPU scheduling have been discussed in the operating systems literature [Coff68, Klei76, Pete86]. Preemptive-resume priority scheduling, where a high-priority job preempts a low-priority job, and the low-priority job resumes its CPU processing after all high-priority jobs have completed their processing, does not seem suitable for database systems. This is because preempting transactions while they hold short-term locks or latches may lead to the convoy problem, as described in [Blas79]. It seems more reasonable, then, to use a non-preemptive form of priority scheduling, while ensuring that high-priority requests do not suffer greatly as a result of the non-preemptability of low-priority requests. In order to reconcile priority with non-preemptability, the CPU can be scheduled according to a priority-based round-robin scheme where the length of a CPU slice is determined by the transaction and not by the scheduler. In this scheme, a transaction gives up the CPU at a "safe point" after a short burst of CPU use. Once the CPU is released, it is assigned to the transaction at the head of the CPU queue. The queue is managed in priority order with requests of the same priority being served in FCFS order.

2.2. DISK SCHEDULING

Classical disk scheduling schemes such as the *shortest seek time first* and *elevator* algorithms [Teor72, Pete86] attempt to minimize the average seek distance. Since the elevator algorithm performs especially well under high disk loads and has good fairness characteristics, it is a good candidate to serve as the basis of a priority scheduling algorithm at the disk. In the elevator algorithm, the disk head is either in an inward-seeking phase or an outward-seeking phase. While seeking inward, it services any requests it passes until there are no more requests ahead. The disk head then changes direction, seeking outward and servicing all requests in that direction as it reaches their tracks.

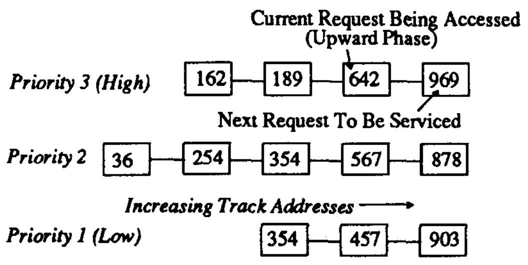
Current Request Being Accessed (Upward Phase)
Priority 3 (High)
162
189
642
969
Next Request To Be Serviced
Priority 2
36
254
354
567
878
Increasing Track Addresses
Priority 1 (Low)
354
457
903

Figure 2.1: Priority-based Disk Scheduling.

In order to support priority, the elevator algorithm can be modified in the following way: disk requests are grouped on the basis of their priority, and the elevator algorithm is used within each group. There is one queue per priority level for buffering outstanding disk requests. Within each queue, requests are arranged in order of their physical (track) addresses. While seeking inward or outward, the disk services any requests that it passes in the currently-served priority queue until there are no more requests ahead. On the completion of each disk request, the scheduler checks to see whether a disk request of a higher priority is waiting for service. If such a request is found, the scheduler switches to the queue that contains the request(s) of the highest priority among those waiting and starts serving that queue. When it switches to a new queue, the request with the shortest seek distance from the head's current position is used to determine the direction in which the head will move.

Figure 2.1 shows the way that the disk scheduler organizes outstanding disk requests in the case of three priority levels. Priority increases as we move from the bottom to the top of the figure. An important side-effect of introducing priority in disk scheduling in this fashion is that the average seek time can worsen as the number of priority levels increases. As an example, consider Figure 2.1, where the disk is currently servicing the request to access track 642 in the queue of priority 3, and the disk head is moving towards higher track numbers. The next request serviced will access track 969, even though there are other requests (for track 878 of priority 2, and track 903 of priority 1) that could be handled earlier as the disk head moves in its upward phase. In the worst case, if each request had a different priority, the prioritized elevator algorithm described above would degenerate into straight priority scheduling. This suggests that in order to provide reasonable I/O performance, it will be preferable to map disk requests into a small number of priority levels at the disk scheduler (even if each transaction has a distinct priority in other parts of the system).

2.3. BUFFER MANAGEMENT

It was shown in [Effe84, Chou85, Sacc86] that the performance of a DBMS can be significantly influenced by the buffer management algorithm used. In general, buffer managers can make three types of decisions: *transaction admission*, *buffer allocation*, and *buffer replacement*. When a transaction arrives at the system, the buffer manager is asked if enough buffer space is available to allow the transaction to be *admitted* into the system.

The arriving transaction may then be blocked outside the system until sufficient memory becomes available. Once a transaction begins running and submits a request for a page that is currently not in the buffer pool, the buffer manager must determine the set of candidate buffers from among which one is to be *allocated* to the transaction. If there are no free buffers, the buffer manager determines which of the data pages currently in the buffer pool should be *replaced*. Buffer management becomes somewhat more complicated when each transaction has a priority, as the buffer demands of different transactions can no longer be treated equally. For example, when making the transaction admission decision, the buffer manager may choose to deallocate buffers from some low-priority transactions in order to allow a high-priority transaction to begin execution. Priorities may also be considered when making replacement and allocation decisions.

The Global LRU buffer management algorithm [Effe84] is simple and is the one most commonly implemented in commercial database systems. The DBMIN algorithm [Chou85], while more difficult to implement, was shown to outperform Global LRU as well as several more sophisticated algorithms (e.g., the "Hot Set" algorithm [Sacc86]). The DBMIN algorithm thus represents the sophisticated end of the buffer management algorithm spectrum, while Global LRU represents the simple, commonly used end. Together, they provide a good pair of algorithms into which the concept of priority may be introduced. In this section, we present two priority buffer management algorithms, the *Priority-LRU* policy, based on Global LRU, and the *Priority-DBMIN* policy, based on DBMIN. As this paper focuses on a query-only environment, both algorithms are described here assuming no updates.[2] *Priority-LRU* and *Priority-DBMIN* differ in their model of the information available to the buffer manager; as a consequence, the buffer pool is organized differently in the two cases. We begin by defining some terms that are needed to describe both algorithms. Then, for each priority-based algorithm, we review the basics of the original algorithm and describe our changes to the buffer pool organization and the policies for transaction admission, buffer replacement, and buffer allocation.

2.3.1. Definitions and Assumptions

At any point in time, the transactions that have accessed a particular resident page (since it was last brought into memory) and are still executing are called the *users* of the page. The *owner* of the page is the transaction with the highest priority among the *users* of that page. A page is *free* if does not have any users. The buffer manager associates a global *timestamp* with each resident page in order to keep track of the recency of usage of pages. A global counter called *bufSequence* is maintained to assign timestamps to resident pages. The *bufSequence* counter is initially set at 0 at system startup time. Each time the data in any of the buffers is accessed, *bufSequence* is incremented and its new value is inserted as the *timestamp* of the data page. Thus, the larger the value of the *timestamp* of a page, the more recently the page was accessed.

A transaction may be admitted to the system right away, or it may be blocked initially by the transaction admission policy. Once a transaction is allowed to begin execution, it continues until it commits or until it is *suspended*. A transaction is said to be *suspended* by the buffer manager if it is temporarily prohibited from making further buffer requests; the buffers owned by the transaction (except those that it has fixed) are freed. The buffer manager considers *reactivating* suspended transactions at the same decision point that it considers admitting blocked transactions, which is whenever a running transaction completes. A reactivated transaction resumes its execution at the point where it was suspended.

2.3.2. The Priority-LRU Algorithm

Global LRU [Effe84] is a buffer replacement policy based on the assumption that there is a temporal locality of data references in relational database operations. Thus, when a buffer frame is required by a transaction, and no free frame is available, the frame with the least recently accessed data (from among all the frames in the buffer pool) should be selected for replacement. The policy is global in that all the frames in the buffer pool are treated according to a single criterion (recency of usage) for allocation as well as for replacement; thus, there is no difference between the policy used for replacement and the policy used for allocation.

Priority-LRU Buffer Pool Organization

In Priority-LRU, our prioritized version of Global LRU, the buffer pool is organized dynamically into priority levels in the following way. At system startup time, all the buffer frames are free, and are arranged in a free list. When a transaction with priority P is allocated a frame from the free list, the frame is inserted in an LRU queue of frames whose owners have priority P. Thus, if there are transactions having m different priority levels at any given time, the buffer pool consists of m LRU queues (one per priority level) and a free list. Figure 2.2 shows an example of the organization of the buffer pool for Priority-LRU; there are three priority levels, and thus three LRU queues, and there are no free frames in this example. Priority increases as we move from the bottom to the top of the figure, and the least recently used page of each queue is in the rightmost frame.

Priority-LRU Transaction Admission

When a new transaction arrives at the DBMS, the buffer manager has to decide whether to allow the transaction to begin execution or not. The key idea in making the admission decision is that, at the very least, every running transaction must have sufficient buffer space to hold all of the pages that it needs to fix concurrently. Otherwise, deadlocks may occur due to contention for buffers. Thus, transactions are required to estimate the maximum number of pages that they will need to fix concurrently, and the buffer manager keeps track of the sum of these "fixing

[2]Update-related buffer management issues (such as when to flush dirty pages to the disk) are essentially orthogonal to the issue of priority-based buffer management. Details of handling updates in a conventional DBMS buffer manager are discussed in [Chen84, Effe84].

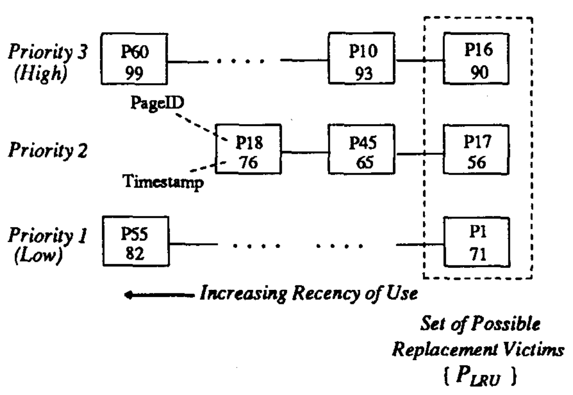

Figure 2.2: Example of Priority-LRU Buffer Pool Organization.

requirements" for all active transactions. If admitting a newly arrived transaction does not cause this sum to exceed the size of the buffer pool, then the transaction is admitted. Otherwise, if there are running transactions of lower priority than the new arrival, the one(s) with the lowest priority among them are suspended until enough buffer space is available for the transaction to be admitted.[3]

Priority-LRU Buffer Replacement and Allocation

As in the conventional Global LRU policy, the buffer allocation and replacement policies coincide in our algorithm. There are two factors that must be weighed when choosing a victim page for replacement: the likelihood that it will be accessed soon, and the priority of the transaction that will access it. These two factors are modeled by the *timestamp* field associated with resident pages and the use of the priority-based LRU queues for buffer pool partitioning, respectively. Note that if priority and recency of usage are the only factors used to choose a replacement victim, then the victim will always be the least recently used page of one of the LRU queues.[4] Let us call the set of possible victims P_{LRU}. In Figure 2.2, P_{LRU} consists of the frames containing the data pages P1, P17, and P16.

The key idea of the replacement policy is that the least recently used page of the lowest priority should be chosen as the victim, with the following caveat: the W_R (for Window of Replacement) most recently accessed data pages should not be chosen for replacement regardless of priority. W_R is a threshold parameter; by varying it, it is possible to vary the relative importance given to recency and priority when making replacement decisions. Let m be the cardinality of P_{LRU}, i.e., let there be m LRU queues in the system. In order to find a replacement victim, we must look at a maximum of m candidates. We start the search at the lowest priority queue, and check whether the candidate's timestamp falls inside the protected W_R window. If it does, we move up one priority level, and we repeat the process until we have either found a victim or else exhausted the search. If all the members of P_{LRU} fall within the window, then the default victim is simply the one with the lowest priority.

An example using Figure 2.2 illustrates the replacement policy. Let *bufSequence*, the global timestamp counter, be 100, and let W_R be 25. We start the search for a victim at P1, the LRU page of the lowest priority level. Since $100-71 > 25$, P1 will be chosen as the victim. If W_R were 30, however, then P1 would fall within the window; in this case, P17 would be chosen as the victim instead.

[3]Among multiple transactions of the same priority, older transactions are favored over younger ones.

[4]Also note that fixed pages cannot be considered for replacement, since this could lead to corruption of data. The admissions policy guarantees that there will always be at least one unfixed replacement candidate.

2.3.3. The Priority-DBMIN Algorithm

As discussed in [Chou85], the primitive operations (e.g., selections, joins) of transactions in a relational DBMS can be described as a composition of a set of regular reference patterns such as sequential scans and hierarchical index lookups. These patterns are known to the query optimizer. The DBMIN buffer management policy makes use of this information using the following key ideas:

(1) Buffers should be allocated to transactions on a "per file instance" basis: i.e., since the pattern of accessing each file used by a transaction can be different, a different set of buffers (called a "locality set") should be allocated to a transaction for each file that it opens.

(2) For each file instance Fi, there is an optimum number of buffers ($OptBufs_{Fi}$) and an optimum replacement policy ($RepPol_{Fi}$). As long as the number of buffers actually allocated to file instance Fi is less than $OptBufs_{Fi}$, the admission policy (see (4)) guarantees that there will be at least one free buffer available for Fi; when the number of buffers allocated to Fi is equal to $OptBufs_{Fi}$, $RepPol_{Fi}$ is used to choose a victim from Fi's locality set when replacement is required. Thus, the replacement policy in DBMIN is local rather than global.

(3) The query optimizer can inform the buffer manager of $OptBufs_{Fi}$ and $RepPol_{Fi}$ for each file instance Fi. The buffer manager can then ensure that the maximum number of buffers allocated to a file instance is $Optbufs_{Fi}$.

(4) The buffer manager ensures that no transaction is allowed to begin running unless it can be guaranteed to get the optimum number of buffers for each of its file instances.

Priority-DBMIN Buffer Pool Organization

As in the original DBMIN algorithm, the buffer pool is organized into "locality sets," where the data pages in each set are all part of the same file instance and have the same *owner*. If a page is accessed by more than one concurrent transaction, its *owner* is the transaction with the highest priority among them. Within a locality set, pages are arranged according to the replacement policy prescribed by the optimizer. The buffer manager maintains the sum of the *OptBufs* values for the running transactions of each priority. Thus, for any priority P, it is easy to compute the sum of the *OptBufs* values for all higher priority transactions that are running.

Priority-DBMIN Transaction Admission

As discussed earlier, DBMIN relies heavily on its Transaction Admission policy. When transactions have priorities, however, the guarantee that any transaction allowed to run will always find *OptBufs* buffers available must be made conditional as follows: Let the combined size of the locality sets of a newly arrived transaction T be $OptBufs_T$, and let its priority be P_T. Let the sum of the optimum sizes of the locality sets of running transactions with priority $\geq P_T$ be $OptBufs_{HIGHER}$, and the sum of the optimum sizes of the locality sets of transactions with priority $< P_T$ be $OptBufs_{LOWER}$. Then, if $(N - OptBufs_{HIGHER}) \geq OptBufs_T$, where N is the total number of buffers in the buffer pool, the buffer manager will admit T. The idea is that if there are sufficient buffers for all currently running transactions of priority P_T or higher and for T itself, then T should be allowed into the system. However, if $(N - OptBufs_{HIGHER} - OptBufs_{LOWER}) < OptBufs_T$, then some buffers may have to be deallocated from lower priority transactions in order to satisfy T's buffer requirements. In this case, the buffer manager will successively suspend transactions (starting with the lowest priority) until there is room enough to admit T.

Priority-DBMIN Buffer Replacement and Allocation

Buffer allocation and replacement are exactly the same as in the original DBMIN algorithm.

3. MODELING A PRIORITY-ORIENTED DBMS

In this section, we describe our performance model of a priority-oriented DBMS. The model, which we implemented using the DeNet simulation language [Livn88], consists of five components: the database itself; a *Source*, which generates the workload of the system; a *Query Manager*, which models the execution behavior of queries; a *Resource Manager*, which models the CPU, I/O, and buffer resources of the system; and a *Concurrency Control Manager*, which implements the details of a particular concurrency control algorithm. Since we will be using only read-only workloads here, we will not discuss the Concurrency Control Manager further.

3.1. MODELING THE DATABASE

The database is modeled as a collection of *relations*. In turn, each relation is modeled as a collection of pages. In addition to relations, the database model contains indices on relations. An index may be either a clustered or a nonclustered B+ Tree. Table 3.1 summarizes the key parameters of the database model. The number of relations in the database is *NumRelations*. For each relation i ($1 \leq i \leq NumRelations$), $RelSize_i$ is the relation size in pages, and $Indexed_i$ determines whether or not the relation has an index. If the relation is indexed, $IndexType_i$ indicates whether the index is clustered or non-clustered, and $Fanout_i$ indicates the fanout of the internal nodes in the index (and thus determines the number of levels of the B+ Tree).

Parameter	Meaning
NumRelations	Number of relations in database
$RelSize_i$	Number of pages in relation i
$Indexed_i$	Whether relation i has an index
$IndexType_i$	Type of index (clustered/nonclustered)
$Fanout_i$	Fanout of internal nodes of index

Table 3.1: Database Model Parameters.

3.2. THE SOURCE MODULE

The Source module is the component responsible for modeling the workload for the DBMS. Table 3.2 summarizes the key parameters of the workload model. A query may belong to any one of *NumClasses* classes, and it may have any one of *NumPriorities* priority levels. The model is that of an open system, and the arrival rate of queries of every <class, priority> combination is controlled by a matrix of arrival rates called *ArrRate*. The arrival of queries of each <class, priority> combination is a Poisson process. Among the per-class parameters is $QueryType_i$, which indicates the query type for the class. Currently, only single-relation select queries and two-relation select-join queries are supported; the model supports selections performed via sequential relation scans, selections performed using either clustered or nonclustered indices, and selections of any of these types followed by a join (which can be either a nested loops join or an index join). Queries are modeled at a logical level because one of the algorithms of interest to us is Priority-DBMIN, which makes use of the logical sequencing of the page accesses of a query. Since the looping behavior of queries can affect buffer hit ratios significantly, the particular query types supported were chosen to provide queries both with and without looping in their page access behavior.

For each query type i, a query plan is provided in the form of a set of parameters. For a join query, $JoinMethod_i$ indicates the join algorithm (nested loops or indexed join), and $Inner_i$ and $Outer_i$ indicate which relation is the inner relation and which is the outer. For each relation j accessed by query i, $AccessPath_{ij}$

Parameter	Meaning
Overall Arrival Pattern Parameters	
NumClasses	Number of query classes
NumPriorities	Number of query priority levels
$ArrRate_{ij}$	Mean exponential arrival rates of queries of class i and priority level j
Per-Class Parameters ($1 \leq i \leq NumClasses$)	
$QueryType_i$	Type of query, e.g., select or select-join
$JoinMethod_i$	Join algorithm used
$Outer_i$	Outer relation
$Inner_i$	Inner relation
$AccessPath_{ij}$	Access path used to access jth relation
$Selectivity_{ij}$	Fraction of jth relation selected
$IndexPageCPU_i$	CPU time for processing an index page
$DataPageCPU_i$	CPU time for processing a data page

Table 3.2: Workload Model Parameters.

indicates the access method used (e.g., a clustered index scan or a sequential scan). In selections, $Selectivity_{ij}$ indicates the proportion of relation j's data that satisfies the selection criterion of query i. We assume a uniform probability of access to all data pages within each relation. The per-class parameter $IndexPageCPU_i$ specifies the expected amount of CPU processing required per index page, and the parameter $DataPageCPU_i$ specifies the expected amount of CPU processing required per data page of each relation accessed for query type i. (The actual CPU processing times per page are exponentially distributed.)

Given a query plan, the Source module generates a list of page accesses that models the sequence in which pages will be accessed by the query. For example, in a selection using a clustered index, the Source uses the selectivity parameter and the size of the target relation to generate a list of page accesses that start at the root of the index, traverse the index to the leaf level, and then access a sequence of index leaf pages and corresponding data pages. In addition to generating a list of page accesses, the Source module also provides information on the number of locality sets, the optimum number of buffers for each set, and the optimal replacement policy (such as MRU or LRU) for each locality set. The maximum number of concurrently fixed pages is similarly provided. This information can be used by the buffer manager. As discussed shortly, the Source must interact with the Resource Manager to determine whether a query is allowed to enter the system right away or whether it has to wait until sufficient buffers become available.

3.3. THE QUERY MANAGER MODULE

The Query Manager is responsible for accepting queries from the Source and modeling their execution. For each page accessed by the query, the Query Manager sends a read request to the Resource Manager; the Resource Manager informs the Query Manager when the read request is completed. The Resource Manager also informs the Query Manager when a query is suspended or reactivated. When the Resource Manager decides to reactivate a suspended query, the Query Manager ensures that the reactivated query resumes execution at the point where it was suspended.

3.4. THE RESOURCE MANAGER MODULE

The Resource Manager controls the physical resources of the DBMS, including the CPU, the disk, and the buffer pool in main memory. Two versions of the Resource Manager have been implemented, supporting the Priority-LRU and Priority-DBMIN buffer management algorithms, respectively. In addition, the model allows priority to be switched on and off at each resource of interest, i.e., at the CPU, the disk, and the buffer pool. ("Switching off" priority at a resource means that all of its service requests are treated as being of equal priority.) The parameters of the Resource Manager are summarized in Table 3.3. *CPUPrio*, *DiskPrio*, and *BufferPrio* are the parameters used to switch priority on and off for the various resources.

CPU and Disk Models

The DBMS has one CPU, which is scheduled using the priority-based round-robin algorithm described in Section 2. The length of each CPU request from a query is its per-page CPU processing time, and each query voluntarily gives up the CPU after processing one page. There is one disk in the system, with requests being scheduled according to the prioritized elevator algorithm of Section 2. Each disk request requires access to one page. The track number of a disk request is chosen at random from among *NumTracks* tracks (i.e., we model the data as being uniformly distributed across all tracks). The total time required to complete a disk access is computed as the sum of its seek time, rotational latency, and transfer time components. The rotational latency and transfer time are together modeled as a single parameter called *DiskConst*. The seek time for seeking across n tracks is computed using the formula:

$$\text{Seek Time}(n) = SeekFactor * \sqrt{n}$$

SeekFactor is specified as a parameter. This square-root relationship between seek time and seek distance is based on the discussion of current disk technology in [Bitt88].

Buffer Manager Models

The buffer manager component of the resource manager encapsulates the details of the buffer management scheme employed by the DBMS. It maintains information about resident pages, and it uses the information provided by the Source module to decide when to allow queries to enter the system. The *NumBuffers* parameter specifies the number of page frames available in the buffer pool. The Priority-LRU and Priority-DBMIN algorithms are each represented by a different buffer manager model, and the Priority-LRU model has an additional parameter W_R that it uses to balance priority and recency when making replacement decisions. In order to simplify the implementation, we decided not to model fixing and unfixing explicitly. However, we do model their effects on query admission decisions in Priority-LRU (in the manner described in Section 2).

4. EXPERIMENTS AND RESULTS

In this section, we present performance results for the priority-oriented DBMS resource scheduling algorithms described earlier. Our goal is to analyze the relative importance

Parameter	Meaning
CPUPrio	Switch to turn priority on/off at CPU
DiskPrio	Switch to turn priority on/off at disk
BufferPrio	Switch to turn priority on/off at buffer pool
NumTracks	Number of tracks per disk
DiskConst	Sum of rotational and transfer delays
SeekFactor	Factor relating seek time to seek distance
NumBuffers	Number of buffer frames in buffer pool
W_R	Window of timestamps used when choosing a replacement victim in Priority-LRU

Table 3.3: Parameters of the Resource Manager.

of priority scheduling at each of the various resources and to understand the interaction between them. In order to simplify the analysis, we consider just two priority levels ("low" versus "high" priority) in the first four experiments presented here. An experiment with four priority levels is also included at the end of this section. Depending on the nature of the workload, any of the physical resources of a DBMS (e.g., the CPU, the disks, or the buffer pool) may become the bottleneck. Together, our experiments cover each of these possibilities.

4.1. Performance Metrics

As discussed earlier, we use an open queuing system to model the DBMS. Response time will thus be the primary performance metric of interest in this study. In particular, we will examine the average response time for queries at each priority level in the workload. Since our performance objective is to provide high priority queries with a preemptive-resume view of the DBMS, we will focus most of our attention on the response time for high priority queries. Two important issues regarding the response time for these queries will be the range over which the system is stable for them and the extent to which the system is able to meet our preemptive-resume performance goal.

Note that in a priority-oriented DBMS, the system can remain stable for high-priority queries long after the arrival rate has become high enough to make the system unstable for low-priority queries. This is because more and more of the system's resources are devoted to high-priority queries as the overall load on the system increases. Thus, there are two regions of operation in each of our experiments. In the first region, the system is stable for both high priority and low priority queries. In the second region, the system has become unstable for low-priority queries, but continues to be stable for high-priority queries. Consequently, we present response time results for low-priority and high-priority queries (in their stable regions) separately for each experiment.

A query's response time is computed by subtracting the time at which the query completes from the time at which it was submitted to the DBMS. As discussed in Section 1, the response time of a query can be broken down into the following components:

$$T_R = T_{W\_EXT} + T_{W\_CPU} + T_{S\_CPU} + N_{DISK} * (T_{W\_DISK} + T_{S\_DISK})$$

We measured each of these components separately in our experiments in order to aid us in analyzing the results.

In order to obtain a statistically significant sample of query response times, each experiment was run long enough for a total of 4000 high-priority queries to complete. (The number of low-priority query completions varied with the load.) Other information was also gathered in the course of each simulation, including the utilization of the CPUs and disks, the average seek time per disk access, and the average number of queries of each priority level running concurrently.

4.2. Parameter Settings

We first present the parameters that were kept constant across all workloads. These parameters are listed in Table 4.1. We then describe our representative workloads and the parameter settings that changed with each workload.

Parameter	Setting
NumRelations	40
RelSize$_i$	1000 pages, 500 pages, 6 pages, 3 pages (10 relations of each size)
Indexed$_i$	YES (1000-page & 500-page relations) NO (6-page & 3-page relations)
IndexType$_i$	Clustered (1000-page & 500-page relations)
Fanout$_i$	20 (1000-page & 500-page relations)
NumTracks	1000
DiskConst	15 ms
SeekFactor	0.6 ms
NumBuffers	50
W_R	10

Table 4.1: Workload-Independent Parameter Settings.

Workload-Independent Parameters

The database is modeled as a collection of 40 relations. We use four different relation sizes in our experiments — 1000 pages, 500 pages, 6 pages, and 3 pages — with the database containing 10 relations of each size. The 1000-page and 500-page relations each have a clustered index available, while the smaller relations are not indexed at all. The disk has 1000 tracks, and the sum of the rotational latency and the transfer time per disk access is 15 milliseconds. The factor relating seek distance to seek time is 0.6 milliseconds[5], so the expected disk access time is between 15 and 30 milliseconds. There are 50 buffer frames in the buffer pool. The W_R parameter used in the Priority-LRU algorithm is set to 10, as this value was found to work well for the range of workloads considered. (For the workloads studied here, the system turns out not to be very sensitive to W_R.)

The model's switches for determining whether or not priorities are used for scheduling at the CPU, the disk, and the buffer pool will be turned off and on as part of each experiment. In the description of the results, the following notation will be used to identify the resources where priority scheduling is turned on: the letters B, C, and D refer to the buffer manager, the CPU, and the disk respectively, while the subscripts *YES* and *NO* refer to priority being turned on and off respectively. For example, the label B_{YES} C_{YES} D_{NO} refers to an experiment where priority is turned on at the buffer manager and at the CPU, but is turned off at the disk.

Workload Parameter Settings

Four different query workloads are employed in our five experiments. Each workload consists of arrival streams of a single query type, but at different levels of priority; in all of the experiments reported here, the arrival rates for each of the priority levels will be equal. The parameters used to generate the four workloads are listed in Table 4.2. The first three workloads

[5]These values for determining disk access times were chosen based on [Bitt88, Gray89].

Parameter	Type I	Type II
QueryType	Select-join	Select-join
JoinMethod	Nested Loops	Nested Loops
Outer (*RelSize*$_1$)	500-page	500-page
Inner (*RelSize*$_2$)	3-page	3-page
AccessPath$_1$	Clustered Index Scan	Clustered Index Scan
AccessPath$_2$	Sequential Scan	Sequential Scan
Selectivity$_1$	1%	1%
IndexPageCPU	15 ms	2 ms
DataPageCPU	15 ms	2 ms
Page Accesses	23	23
Locality Set Sizes (*index*, *outer*, *inner*)	1, 1, 3	1, 1, 3
Replacement Policies (*index*, *outer*, *inner*)	MRU, MRU, MRU	MRU, MRU, MRU
Fixing Requirements	3	3
Parameter	**Type III**	**Type IV**
QueryType	Select-join	Select
JoinMethod	Nested Loops	-
Outer (*RelSize*$_1$)	500-page	1000-page
Inner (*RelSize*$_2$)	6-page	-
AccessPath$_1$	Clustered Index Scan	Clustered Index Scan
AccessPath$_2$	Sequential Scan	-
Selectivity$_1$	1%	1%
IndexPageCPU	2 ms	5 ms
DataPageCPU	2 ms	5 ms
Page Accesses	38	13
Locality Set Sizes (*index*, *outer*, *inner*)	1, 1, 6	1, 1, -
Replacement Policies (*index*, *outer*, *inner*)	MRU, MRU, MRU	MRU, MRU, -
Fixing Requirements	3	2

Table 4.2: Workload Parameter Settings.

consist of select-join queries, with the result of a selection using a clustered index on a 500-page outer relation being joined to a smaller inner relation; the fourth workload consists of a clustered index selection on a 1000-page relation. For each relation accessed by a query, the actual relation accessed was chosen uniformly from among the 10 relations of that size, so there is not much data sharing in the system. To aid in understanding the nature of the workloads, Table 4.2 also includes the total number of page accesses for each query type. In addition, it lists the optimum locality set sizes and replacement policies for Priority-DBMIN, and it lists the query fixing requirements for Priority-LRU.

4.3. Experiment I (Admission Control)

In this experiment, we investigate the impact of priority scheduling on performance under the Type I workload. This workload consists of a mix of CPU-intensive, indexed-select/nested-loops-join queries. In fact, given the Type I parameter settings, the disk utilization is only about 65% when the CPU is fully utilized, so the presence or absence of priority-based disk scheduling has no effect on performance. Thus, the priority scheduling combinations studied here are those with and without CPU and buffering priority (i.e., $B_{NO}C_{NO}D_{YES}$, $B_{NO}C_{YES}D_{YES}$, $B_{YES}C_{NO}D_{YES}$, and $B_{YES}C_{YES}D_{YES}$).

Figure 4.1 shows the response time results for high priority queries using the Priority-DBMIN buffering algorithm, and Figure 4.2 shows the corresponding results for low priority queries. The arrival rate axis of these figures indicates the *combined arrival rate* of both priority levels. The arrival rates for the two priority levels are equal, so the arrival rate for each of the levels is half of this total rate. In addition to the curves showing the high priority response times for the four scheduling combinations, Figure 4.1 also includes a curve labeled HPO (for High-Priority Only) that indicates what the high priority response times would be if no low priority queries were arriving. This curve will help us evaluate how successful we are at getting the DBMS to treat high priority queries like a preemptive-resume server should; the better we do at approximating this curve, the closer we are to meeting this design goal. Figure 4.2 contains a similar curve (labeled LPO) that shows how the system would perform with low priority queries only, indicating the extent to which low priority queries suffer due to competition from high priority queries. Despite the fact that the system is CPU-bound, Figures 4.1 and 4.2 show that the use of priority in the buffer manager is extremely important. Without priority in the buffer manager, the system saturates at an arrival rate of about three queries per second for both low and high priority queries. With priority, however, the system is stable for high priority queries until the arrival rate reaches about six queries per second. These differences, as we will see, are due to the use of priority in the Priority-DBMIN admission control policy.

To understand why admission control has such an effect, let us consider how priority affects the view that queries of each priority have of a given resource. Arriving low priority queries see a resource containing other queries of both low and high priority. This view is the same whether or not priority scheduling is employed at the resource, although the quality of service that low priority queries receive is affected by the scheduling policy. This is consistent with Figure 4.2, which shows that low priority query performance is affected only slightly by the presence or absence of priority scheduling. The system always saturates at the same point from the perspective of low priority queries. High priority queries get the same view of a resource as low priority queries if it is not scheduled using priority. This explains why, when the buffer manager does not use priority to control admission in Figure 4.1 (in the B_{NO} curves), the system saturates at three queries per second for high priority queries. In this case the saturated "resource" is the DBMS itself, and if priority is not used for admission control, the external waiting time for high priority queries is limited by the response time of low priority queries. However, when the buffer manager's admission policy does favor high priority queries, the DBMS admits them preferentially and even suspends low priority queries in their favor. In this case, they see only other high priority queries in the system, so they essentially see only half of the actual system load. This

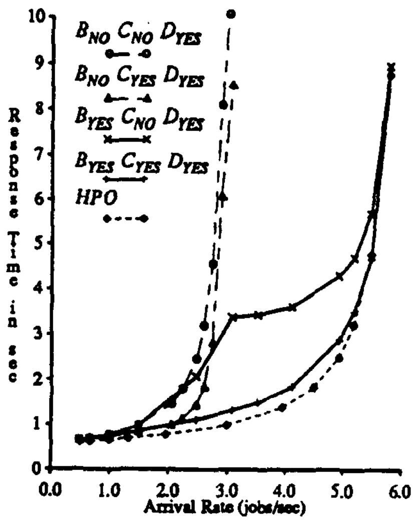

Figure 4.1: Type I: High Priority. (Priority-DBMIN)

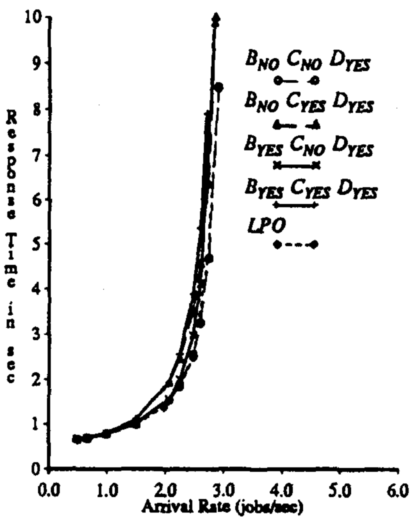

Figure 4.2: Type I: Low Priority. (Priority-DBMIN)

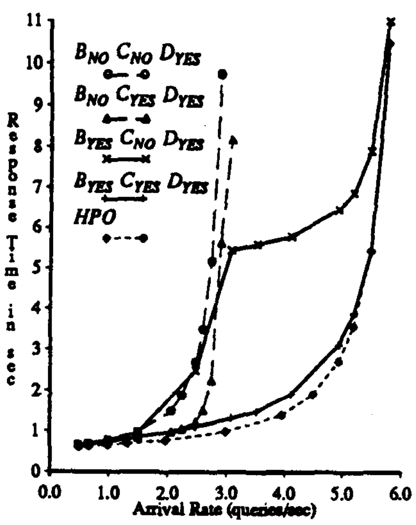

Figure 4.3: Type I: High Priority. (Priority-LRU)

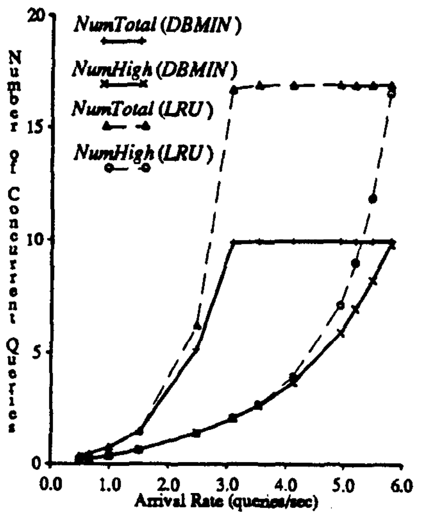

Figure 4.4: Number of Concurrent Queries.

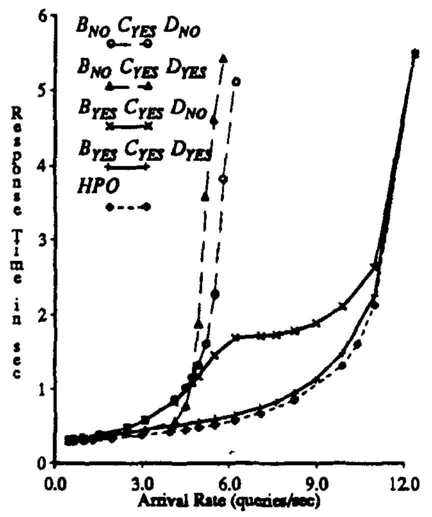

Figure 4.5: Type II: High Priority. (Priority-DBMIN)

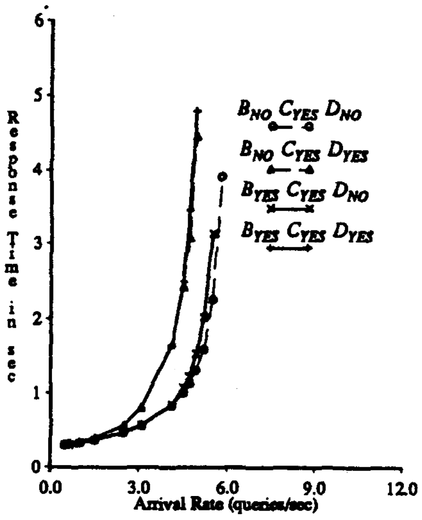

Figure 4.6: Type II: Low Priority. (Priority-DBMIN)

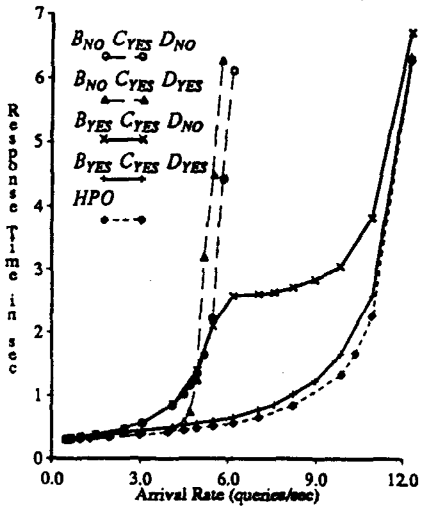

Figure 4.7: Type II: High Priority. (Priority-LRU)

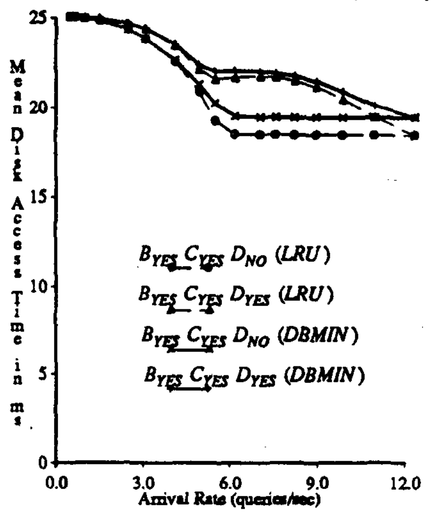

Figure 4.8: Mean Disk Access Times.

explains why the system remains stable twice as long for high priority queries under Priority-DBMIN.

Another interesting feature of Figure 4.1 is how priority scheduling at the CPU affects the response time for high priority queries under Priority-DBMIN. With priority used in the buffer manager but not at the CPU (i.e., the $B_{YES} C_{NO} D_{YES}$ curve), high priority response time is similar to low priority response time until the system saturates for low priority queries. After this point, the response time is stable for a while as more and more low priority queries are suspended due to high priority arrivals. That is, the buffer manager has already admitted as many queries as it can, so all it can do in this range is trade low priority queries for high priority ones. Since the workload is CPU-bound, the load that high priority queries see does not change in this range if CPU scheduling is not priority-based. The CPU eventually becomes saturated with high priority queries alone (at an arrival rate of six queries per second, which is three high priority queries per second). However, when CPU scheduling is also priority-based in Figure 4.1 (i.e., the $B_{YES} C_{YES} D_{YES}$ curve), we see a different trend. Here, the high priority response time behavior is very close to that of the HPO curve; recall that this curve shows the response time behavior without low priority queries. This indicates that priority scheduling at the CPU is succeeding at shielding the high priority queries from the low priority queries (i.e., giving them the impression of being the only queries in the system). Thus, a combination of Priority-DBMIN and priority CPU scheduling indeed gives high priority queries a preemptive-resume-like view of the DBMS.

Figure 4.3 shows the response time results for high priority queries using the Priority-LRU buffering algorithm; we do not show the low priority query results, as they are very similar to those for Priority-DBMIN in Figure 4.2. The Priority-LRU trends in Figure 4.3 are quite similar to those that we saw in Figure 4.1, and the explanation of the trends are the same here. The main difference between the Priority-LRU and Priority-DBMIN results is that priority CPU scheduling is more critical for Priority-LRU. The two algorithms perform almost identically when priority scheduling is used everywhere (except at the highest loads, where Priority-DBMIN is marginally better). However, when CPU scheduling is not priority-based, high priority queries suffer more under the Priority-LRU algorithm. This is due to the difference between the admission control policies of the two algorithms. Since the Priority-DBMIN policy is more conservative, it admits fewer queries than Priority-LRU does. This is evident in Figure 4.4, which shows the number of concurrent queries (both high priority and total) inside the system, as opposed to waiting outside, under the two buffer management algorithms. Priority scheduling at the CPU changes the number of competing queries seen by high priority queries from being the total number in the system to being just the high priority number, which is a bigger change in the Priority-LRU case. The general Priority-LRU conclusions are the same, though: priority is needed both in the buffer manager (for priority-based admissions control) and at the CPU in order to achieve the HPO performance objective for high priority queries.

4.4. Experiment II (Disk Priority)

In our second experiment, we focus on the impact and trade-offs involved in priority scheduling at the disk. Here we use the Type II workload, which is just like the Type I workload except that the per-page CPU time for queries is now just 2 milliseconds instead of 15 milliseconds. Thus, the workload consists of a mix of I/O-intensive, indexed-select/nested-loops-join queries. With this workload, the CPU utilization is approximately 35% when the disk becomes 100% utilized, so the presence or absence of priority-based CPU scheduling has no effect here. Thus, we examine situations with priority scheduling at the disk and the buffer manager (i.e., $B_{NO} C_{YES} D_{NO}$, $B_{NO} C_{YES} D_{YES}$, $B_{YES} C_{YES} D_{NO}$, and $B_{YES} C_{YES} D_{YES}$).

Figure 4.5 shows the response time results for high priority queries using Priority-DBMIN, and Figure 4.6 shows the corresponding results for low priority queries. Figure 4.7 presents the high priority query response time results for Priority-LRU; again, the low priority results for Priority-LRU were very much like those of Priority-DBMIN, so we omit them here. Figure 4.8 shows the mean disk access times for Priority-DBMIN and Priority-LRU with and without priority-based disk scheduling (for the two cases $B_{YES} C_{YES} D_{NO}$ and $B_{YES} C_{YES} D_{YES}$). For the most part, Figures 4.5-4.7 display the same trends that we saw in Figures 4.1-4.3, and they do so for the same reasons (albeit with a different bottleneck resource). Again, it is evident that priority must be incorporated in the buffer manager as well as the bottleneck resource (the disk) in order to provide the desired level of performance for high priority queries; and again, priority scheduling of the bottleneck resource is more important for Priority-LRU because of its less effective admission control policy. However, there are also several other interesting points to be noted from this experiment.

The first point to notice is that in Figure 4.6, unlike Figure 4.2, low priority queries clearly suffer performance-wise due to priority-based scheduling of the bottleneck resource (the disk in this case). This is due to the fact that, in the range where they suffer, priority scheduling has reduced the service capacity of the disk by increasing the mean disk access time (as shown in Figure 4.8). Combined with large waiting times, which are in excess of eight times larger than those for high priority queries, this produces earlier response time degradation for low priority queries. In contrast, high priority response time is again close to that of the HPO curve when priority scheduling is used everywhere, indicating that high priority queries are largely unaffected by the somewhat increased mean disk access time. This is because this increase is more than offset by the decrease in disk waiting times that priority-based disk scheduling produces for the high priority queries. Put another way, there is a price to be paid for doing disk scheduling based on priority, but it is the low priority queries that pay the price.

Another interesting observation can be made from Figure 4.8. At low arrival rates, where the disk load is low, there is no penalty for priority-based disk scheduling. This is because there is little or no queuing for the disk in this region. However, there is also no penalty at the highest arrival rates. The explanation is different in this case. Here, the reason that priority scheduling is able to do as well as the strict elevator algorithm is that it, too, effectively becomes the elevator algorithm; the disk is so heavily loaded due to high priority requests that it is kept busy serving their elevator queue. In the middle range of arrival rates, where there is a mean disk access time penalty, the benefits of priority scheduling far outweigh the penalty for high priority queries, as described above. Thus, priority-based disk scheduling appears to be very worthwhile for a priority-based DBMS.

4.5. Experiment III (Buffer Priority)

In our third experiment, we focus on the relative behavior of the DBMIN-based and LRU-based approaches to buffer management. Here we use the Type III workload, which is similar to the Type II workload but with an inner relation that is twice as large as before. Thus, the queries in the workload have a more significant looping behavior here. Again, we focus our attention on priority scheduling at the disk and in the buffer manager.

Figure 4.9 shows the response time results for high priority queries using Priority-DBMIN, and Figure 4.10 shows the corresponding results for low priority queries. The trends here are basically those of Experiment II, but the absolute performance is different due to the larger query size. The only apparent relative difference is that priority disk scheduling leads to somewhat less of an improvement in the high priority response time when buffering priority is used. This is because the bottleneck here is actually buffer space; the disk utilization never exceeds 90% because the admission control policy is unable to allow enough queries into the system to saturate the disk.

Figure 4.11 presents the high priority query response time results for Priority-LRU; as always, the low priority results for Priority-LRU are virtually identical to those for Priority-DBMIN, so we do not show them. For the high priority queries, we see significantly different behavior for Priority-LRU here (relative to Priority-DBMIN) than that observed in previous experiments, especially without disk priority scheduling. The difference is a consequence of the admission control policies of the two algorithms. As was shown in [Chou85], the information-based admission controller of the basic DBMIN algorithm can significantly reduce thrashing relative to the basic LRU algorithm, and the same is (of course) true of our priority-based versions of these algorithms. Figure 4.12 shows the buffer pool hit ratio for high priority queries under both the Priority-DBMIN and Priority-LRU algorithms. The hit ratio for Priority-LRU drops significantly where the high priority response times increase in Figure 4.11, indicating that this is indeed the problem. This thrashing is due to the fact that Priority-LRU admits too many high priority queries, which leads them to take buffers from one another since the number of available low priority buffers is insufficient. The result is especially drastic without priority at the disk. The low priority queries in the system have extremely low hit ratios, as their buffers are consistently chosen as replacement victims for high priority queries; this causes them to generate many more disk requests than they would under the carefully controlled Priority-DBMIN policy. Without priority at the disk, this volume of requests generates much more disk traffic for high priority queries to contend with, increasing their response times. Buffering priority alone is thus not enough to ensure stability for high priority queries in this workload under Priority-LRU.

4.6. Experiment IV (Minimal Buffer Contention)

In our final two-priority experiment, we investigate the performance of the priority-based scheduling algorithms for a workload where buffer contention is not a significant factor. Here we use the Type IV workload, which consists of indexed-selection queries with no looping behavior. The Type IV workload is I/O-bound due to its 5 millisecond page CPU time and lack of looping behavior, so we again focus on priority scheduling at the disk and in the buffer manager.

Figure 4.13 shows the response time results for high priority queries using Priority-DBMIN, and Figure 4.14 shows these results for Priority-LRU. We omit the low priority results, as they are qualitatively the same as in Experiments II and III. The main things to observe here are the impact of priority-based management of the disk and buffer resources on the performance of high priority queries. Since buffer contention is a non-issue under this workload, the Priority-DBMIN and Priority-LRU algorithms perform pretty much identically. However, they are still essential: without a priority-based buffer management algorithm (for priority-based admission control), the system saturates for high-priority queries at 4.5 queries per second; with priority in the buffer manager, the saturation point is extended to nine queries per second. Likewise, priority disk scheduling is crucial for good performance here. This is because the workload is I/O bound and buffer hits are rather rare.

4.7. Experiment V (Four Priorities)

In the last experiment of the paper, we briefly examine the performance of the priority-based scheduling algorithms for a workload consisting of four levels of priority. We return to the Type III (buffer-intensive, I/O bound) queries for this experiment. Here we look only at how queries of the different priority levels perform when priority is employed everywhere (i.e, at the CPU scheduler, the disk scheduler, and the buffer manager).

Figure 4.15 shows the query response time results for the four priority levels under Priority-DBMIN, and Figure 4.16 shows the results for the Priority-LRU algorithm. The observed trends are what we would expect based on the results of our earlier experiments. First, the system succeeds at providing preemptive-resume-like performance, as the highest priority query response times are quite close to the HPO curves. Second, the saturation points are not linearly distributed over the range of arrival rates. This is because of the system view that priority scheduling provides to the different priority levels: the highest priority queries see only themselves (i.e., one-fourth of the load), queries at the second-highest priority see themselves and the highest priority queries (i.e., one-half of the load), the second-lowest priority queries see the top three levels (i.e., three-fourths of the load), and the lowest priority queries see the entire query load as competition for system resources. Finally, as in Experiment III, Priority-DBMIN provides somewhat better performance than Priority-LRU here, as the workload is one where buffer management has a significant role to play in determining performance.

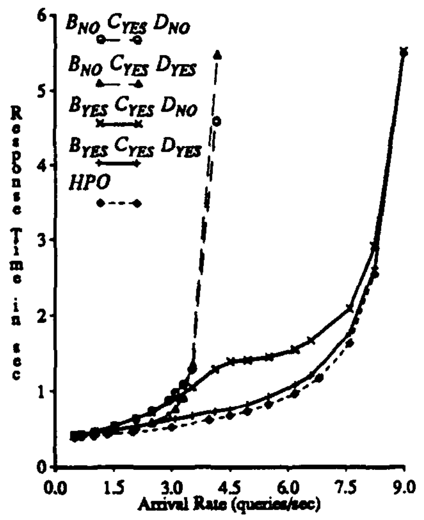

Figure 4.9: Type III: High Priority. (Priority-DBMIN)

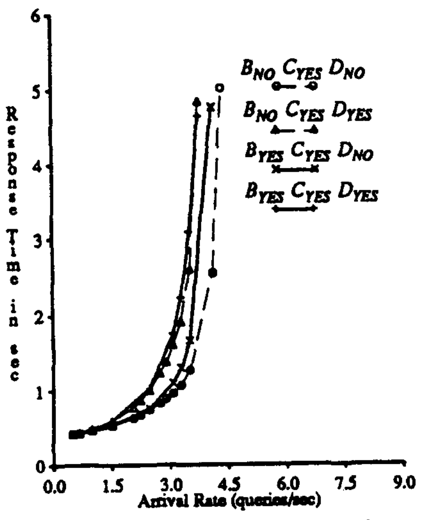

Figure 4.10: Type III: Low Priority. (Priority-DBMIN)

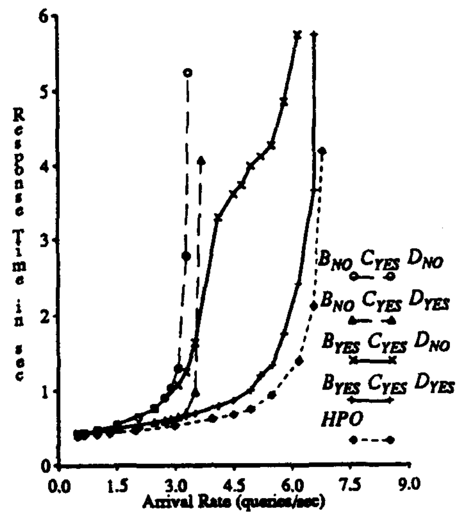

Figure 4.11: Type III: High Priority. (Priority-LRU)

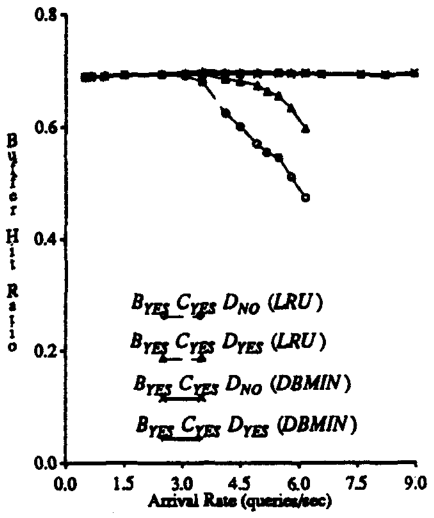

Figure 4.12: Buffer Hit Ratio. (High Priority)

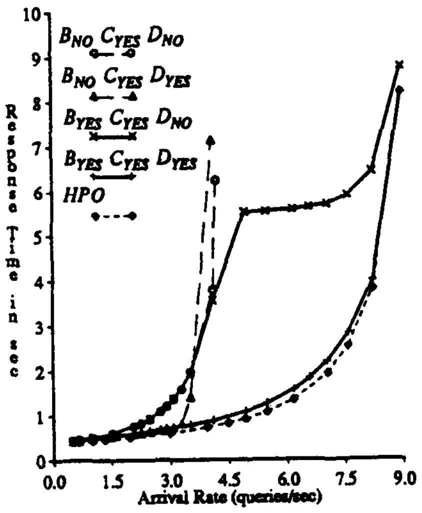

Figure 4.13: Type IV: High Priority. (Priority-DBMIN)

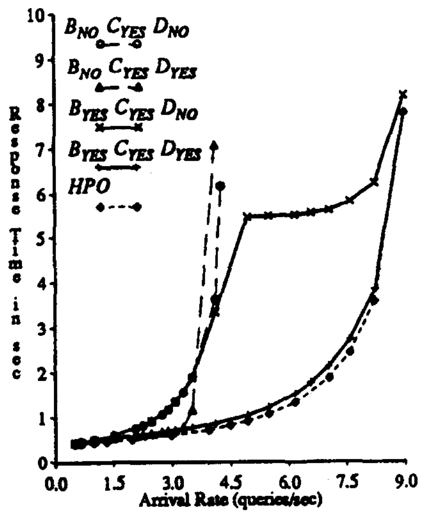

Figure 4.14: Type IV: High Priority. (Priority-LRU)

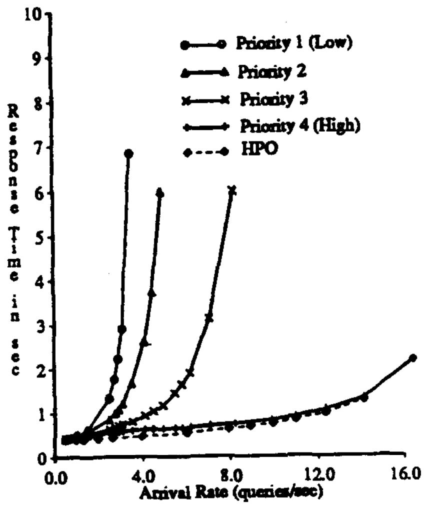

Figure 4.15: Four Priority Levels (Priority-DBMIN)

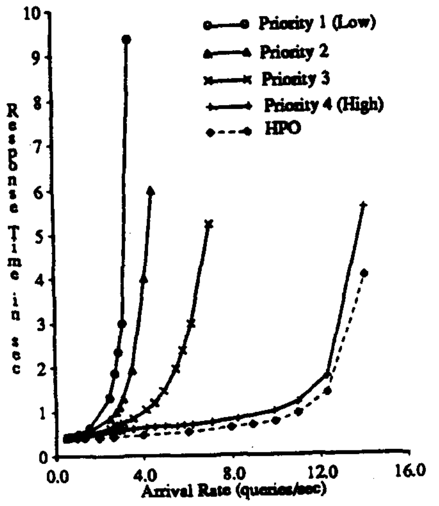

Figure 4.16: Four Priority Levels (Priority-LRU)

5. CONCLUSIONS AND FUTURE WORK

This paper has examined the problem of priority scheduling in a database management system. First, the architectural consequences of adding priority to a DBMS were investigated, given the importance of effectively scheduling the multiple, heterogeneous resources of such a system. Several concrete, priority-based algorithms were then proposed for managing DBMS resources, including a priority-based disk scheduling algorithm and two algorithms for priority-based buffer management. The proposed disk scheduling algorithm is a priority-based variant of the elevator algorithm, and the two buffer management proposals are extensions of the LRU and DBMIN algorithms.

In addition to suggesting approaches to DBMS priority scheduling, we studied their performance through simulation and obtained a number of interesting performance insights. Using preemptive-resume as our model of desirable priority scheduling behavior, we found that it is indeed possible to do a good job of priority scheduling in a DBMS context. However, our simulation results indicate that the objectives of priority scheduling cannot be met by a single priority-based scheduler. Rather, whether the system bottleneck is the CPU or the disk, it is essential that priority scheduling on the critical resource be used in conjunction with a priority-based buffer management algorithm. Between our two proposed algorithms, we found that Priority-DBMIN dominates Priority-LRU in cases where buffer contention is a factor. When buffer contention is not a key factor, either algorithm is sufficient to enable the system to achieve its performance goals.

We view this work as a first step, with a number of interesting problems and opportunities for future work remaining. First, in the area of algorithm design, there are a number of alternative design decisions that we could have made differently in the Priority-DBMIN and Priority-LRU algorithms; we plan to study these design tradeoffs carefully in the future. Second, our performance study focused on a read-only workload. While we believe that allowing high priority queries to be update queries would not alter our conclusions significantly, we do need to examine the interaction of lower priority updates and the admission control policy. One issue here is that suspending an update query and replacing its pages will carry the price of writing its updates to disk. A related issue is the impact of suspending queries that hold locks; restarts might be superior to suspensions in such situations. Finally, at a higher level, we plan to study how priority can be used as a means to meeting performance goals in a DBMS context. In particular, we plan to study the problem of mapping soft real-time constraints into priorities in such a way as to minimize missed deadlines.

REFERENCES

[Abbo88] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Bitt88] Bitton, D., and Gray, J., "Disk Shadowing," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Blas79] Blasgen, M., et al, "The Convoy Phenomenon," *Operating Sys. Rev.* 13(2), April 1979.

[Buch89] Buchmann, A., et al, "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *Proc. 5th Data Eng. Conf.*, Los Angeles, CA, Feb. 1989.

[Chan85] Chang, H.-Y., and Livny, M., "Priority in Distributed Systems," *Proc. IEEE Real-Time Sys. Symp.*, Dec. 1985.

[Chan87] Chang, H.-Y., *Dynamic Scheduling Algorithms for Distributed Soft Real-Time Systems*, Ph.D. Thesis, Comp. Sci. Dept., Univ. of Wisconsin-Madison, Sept. 1987.

[Chen84] Cheng, J., et al, "IBM Database 2 Performance: Design, Implementation, and Tuning," *IBM Sys. J.* 23(2), 1984.

[Chou85] Chou, H-T., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. 11th VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[Coff68] Coffman, E., and Kleinrock, L., "Computer Scheduling Methods and Their Countermeasures," *Proc. AFIPS Spring Joint Comp. Conf.*, April 1968.

[Daya88] Dayal, U., et al, "HiPAC: A Research Project in Active, Time-Constrained Database Management," Technical Report CCA-88-02, Computer Corporation of America, Boston, June 1988.

[Effe84] Effelsberg, W., and Haerder, T., "Principles of Database Buffer Management," *ACM Trans. on Database Sys.* 9(4), Dec. 1984.

[Gray89] Gray, J., personal commmunication.

[Jens86] Jensen, E. D., Locke, C. D., and Tokuda, H., "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proc. IEEE Real-Time Sys. Symp.*, Dec. 1986.

[Klei76] Kleinrock, L., *Queueing Systems*, John Wiley and Sons, 1976.

[Livn88] Livny, M., *DeNet User's Guide*, Version 1.0, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1988.

[Pete86] Peterson, J., and Silberschatz, A., *Operating Systems Concepts*, Addison-Wesley, 1986.

[SIGM88] *SIGMOD Record* 17(1), Special Issue on Real-Time Data Base Systems, S. Son, ed., March 1988.

[Sacc86] Sacco, G.M., and Schkolnick, M., "Buffer Management in Relational Database Systems," *ACM Trans. on Database Sys.*, 11(4), Dec. 1986.

[Stan88] Stankovic, J., "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Computer* 21(10), Oct. 1988.

[Teor72] Teorey, T., and Pinkerton, T., "A Comparative Analysis of Disk Scheduling Policies," *Comm. ACM* 15(3), March 1972.

[Wede86] Wedekind, H., and Zoemtlein, G., "Prefetching in Realtime Database Applications," *Proc. 1986 SIGMOD Conf.*, Washington, D.C., June 1986.