Query Processing in Firm Real-Time Database Systems

Hwee Hwa Pang

Technical Report #1221

March 1994

QUERY PROCESSING IN FIRM REAL-TIME DATABASE SYSTEMS

by

HWEE HWA PANG

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1994

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 as a research area. The basic mechanism that all real-time systems employ to enable jobs to meet their deadlines is priority scheduling. The adoption of priority scheduling requires changes in the ways that database systems service their jobs, which are traditionally not priority-based. In this thesis, we propose and evaluate techniques to allow queries to execute efficiently in priority scheduling environments such as those envisioned for real-time database systems.

One of the key system resources for processing queries efficiently is memory (i.e., buffer pages). To maximize the effectiveness of priority scheduling, it must be possible for running low-priority queries to relinquish buffers for use by higher-priority queries that just arrived at the system. In addition, when these higher-priority queries leave the system, freeing up memory, it would help the low-priority queries to meet their deadlines if they could accept and make good use of the additional memory during the remainder of their execution. Due to their heavy reliance on main memory, queries performing hash joins and external sorts are especially vulnerable to fluctuations in their memory allocations. The first part of this thesis focuses on the design and evaluation of memory-adaptive query primitives that allow hash joins and external sorts to execute efficiently in the face of memory fluctuations.

With the low-level query primitives in place, the second part of the thesis is devoted to addressing higher-level query scheduling issues, including admission control, memory allocation, and priority assignment. In particular, an algorithm is proposed that dynamically controls the multiprogramming level and the memory allocation strategy for queries in a real-time database system to balance the demands on its memory, CPUs, and disks. Through a series of simulation experiments, this algorithm is shown to perform well over a wide range of workloads, but to be biased when presented with a multiclass workload. The algorithm is then augmented with a class-priority adaptation mechanism to enable it to better handle the performance demands of multiclass query workloads. The augmented algorithm strives not only to minimize the number of missed deadlines, but also to distribute any missed deadlines among all query classes according to a set of administratively-defined performance objectives.

ACKNOWLEDGEMENTS

I have been very fortunate to have both Professor Miron Livny and Professor Michael Carey as my thesis advisors. Miron deserves credit for introducing me to real-time database research. During my graduate career, he has repeatedly challenged me to strive for excellence in my work. I shall fondly remember the many hours of discussion we had over the past three years, especially that "memorable" session in my living room. Mike is one of the most giving persons I have ever met. As my mentor, he has been a constant source of support, encouragement, and sound advice. He has also provided me with invaluable technical guidance, steering me back in the right direction on those occasions when I have wondered off course. Despite my incessant hounding, he has never lost his patience. I have enjoyed working with him tremendously. I salute these two great guys!

I am also grateful to the other members of my committee, Professors David DeWitt, Jeffrey Naughton, and Parmesh Ramanathan, for their constructive criticisms and suggestions. David, in particular, has helped me revise the experiments in this thesis.

During my stint in Madison, I have benefited from my interaction with Paul Bober, Kurt Brown, Michael Cheng, Mike Franklin, Manish Mehta, C.K. Tan, and other past and present graduate students at UW.

The National University of Singapore has been very generous in giving me a scholarship for the entire duration of my Ph.D. studies. I would like to express my appreciation for Dr. Desai Narasimhalu, my manager at the Institute of Systems Science, and Dr. Juzar Motiwalla, the director of the Institute of Systems Science, for their faith in me and for recommending me for the scholarship.

Since young, my parents have impressed upon me the importance of getting a good education. My family's support of my decision to pursue a Ph.D. meant a great deal to me. I have been blessed with a marvelous wife, Janice, who made a huge sacrifice in leaving her job to be with me. The few holiday trips that we have taken here are some of the happiest times in my life. Janice has been extraordinarily understanding when I needed to work long hours to meet paper submission deadlines, though she also has consistently plotted to foul my attempts to work 25 hours a day. Without her steadfast love and encouragement, I would never be able to go through this demanding program. Truly, "whoso findeth a wife findeth a good thing, and obtaineth favor from the Lord" (Proverbs 18:22).

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iv
TABLE OF CONTENTS vi
Chapter 1: INTRODUCTION 1
1.1 Real-Time Database System Architecture 3
1.2 Thesis Contributions 5
1.2.1 Memory-Adaptive Query Processing 6
1.2.2 Admission Control and Memory Allocation 7
1.3 Organization of the Thesis 8
Chapter 2: MODEL AND METHODOLOGY 9
2.1 Database System Simulation Model 9
2.1.1 Database Module 10
2.1.2 Source Module 11
2.1.3 Query Manager 12
2.1.4 Memory Manager 14
2.1.5 CPU and Disk Managers 15
2.1.6 Concurrency Control Manager 16
2.2 Experimental Methodology and Performance Metrics 16
Chapter 3: PARTIALLY PREEMPTIBLE HASH JOINS 19
3.1 Related Work on Hash Joins 20
3.2 Memory-Adaptive Hash Join Algorithms 22
3.2.1 Partially Preemptible Hash Join 23
3.2.1.1 PPHJ: The Basics 24
3.2.1.2 PPHJ: Variations on a Theme 25
3.2.2 Other Algorithms 27
3.2.2.1 Nakayama et al 27
3.2.2.2 Zeller and Gray 27
3.2.2.3 GRACE and Hybrid 28
3.3 Experiments and Results 29
3.3.1 Source of Memory Fluctuations 29
3.3.2 Baseline Experiment 30
3.3.3 Memory Contention 35
3.3.4 M to $||R||$ Ratio and $||S||$ to $||R||$ Ratio 38

3.3.5 Magnitude of Memory Fluctuations 40
3.3.6 Rate of Memory Fluctuations 42
3.3.7 Discussion of Other Alternatives 45
3.4 Conclusion 46
Chapter 4: MEMORY-ADAPTIVE EXTERNAL SORTING 49
4.1 Standard External Sort Algorithms 50
4.1.1 In-Memory Sorting Methods 50
4.1.2 Merging Strategies 51
4.2 Memory-Adaptive External Sort Algorithms 53
4.2.1 Split-Phase Adaptation 53
4.2.2 Merge-Phase Adaptation Strategies 54
4.2.2.1 Suspension 54
4.2.2.2 Paging 54
4.2.2.3 Dynamic Splitting 55
4.2.3 Notation for External Sort Algorithms 58
4.3 Experiments and Results 59
4.3.1 Source of Memory Fluctuations 59
4.3.2 No Memory Fluctuation 60
4.3.3 Baseline Experiment 63
4.3.4 M to $||R||$ Ratio 67
4.3.5 Magnitude of Memory Fluctuations 70
4.3.6 Rate of Memory Fluctuations 72
4.4 Sort-Merge Joins 74
4.4.1 Memory-Adaptive Sort-Merge Joins 74
4.4.2 Experiment and Results 75
4.5 Conclusion 76
Chapter 5: MANAGING MEMORY FOR REAL-TIME QUERIES 79
5.1 Related Scheduling Work 80
5.2 Priority Memory Management 80
5.2.1 Admission Control 81
5.2.1.1 Miss Ratio Projection 82
5.2.1.2 Resource Utilization Heuristic 83
5.2.2 Memory Allocation 84
5.2.3 Dealing with Workload Changes 87
5.2.4 An Example 87
5.3 Experiments and Results 88
5.3.1 Baseline Experiment 89

5.3.2 Moderate Disk Contention 95
5.3.3 Workload Changes 98
5.3.4 Desirable Resource Utilization Levels 100
5.3.5 Other Query Types 101
5.3.6 Multiclass Workload 103
5.3.7 Scalability of Results 103
5.4 Conclusion 105
Chapter 6: MULTICLASS QUERY SCHEDULING 107
6.1 Priority Query Scheduling 108
6.1.1 Priority Memory Management for Multiclass Workloads 108
6.1.2 Priority Adaptation Query Scheduling 110
6.2 Experiments and Results 113
6.2.1 Baseline Experiment 114
6.2.2 Skewed Class Objectives 120
6.2.3 Identical Classes 123
6.2.4 Workload Changes 125
6.2.5 Three-Class Workloads 127
6.3 Conclusion 129
Chapter 7: CONCLUSIONS 130
7.1 Memory-Adaptive Query Processing Results 130
7.2 Real-Time Query Scheduling Results 131
7.3 Future Work 132

CHAPTER 1

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. Moreover, these applications may generate jobs that have to be completed by certain deadlines for the results to be of value in providing decision-support [Abbo88a, Stan88]. For example, in a factory automation system, a quality inspection must be completed within a specified time frame in order to undertake any necessary corrective actions. The need for systems that are able to manage substantial amount of data in a timely fashion has contributed to the emergence of real-time databases (RTDBS) as a research area.

The area of real-time database management is the result of a confluence between two previously separate areas — database management systems and real-time computing. On one hand, while database management systems provide efficient mechanisms for querying vast quantities of shared data, the notion of deadline is foreign to these systems. On the other hand, past research on real-time computing has focused on developing systems that support the time constraints that are native to real-time applications. To date, a number of real-time task scheduling policies have been proposed, and numerous research results have been obtained for both uniprocessor and multiprocessor systems [Liu73, Dert74, Mok78, Jens85, Lock86, Panw88, Baru91]. However, these studies have ignored the need to manage substantial amounts of data. RTDBSs are designed to combine the strengths of these two areas, i.e., to provide timely retrievals and updates for real-time applications that require access to shared data that resides on secondary storage.

In a non-real-time database system, the usual performance objective is to minimize the mean response time or to maximize the system throughput. For such a system, the primary concern is the overall system performance, rather than the response time of individual jobs. In the case of an RTDBS, however, it is often desirable to manipulate the response time of individual jobs because doing so *can* lead to fewer deadline misses, i.e., better performance. To illustrate this point, suppose that two jobs with different timing requirements are submitted to an RTDBS. If the system uses a round-robin resource scheduling policy to service the two jobs, it may provide timely services to the less urgent job but miss the deadline of the more urgent job. Instead, by giving preferential service to the job whose deadline is more imminent, the system may be able to complete both jobs on time. In order to regulate the progress of individual jobs, RTDBSs prioritize their jobs and schedule them according to their priorities.

To ensure the effectiveness of priority scheduling, *all* of the resource schedulers in an RTDBS must be priority-driven. It is inadequate for the scheduling decisions of only some of the resources to be priority-cognizant because this leads to the possibility of a low-priority job blocking the progress of a higher-priority job by depriving the latter of resources that are not priority-scheduled. Such a situation, known as priority inversion, causes priority scheduling to be counterproductive to system performance [Sha90]. For example, if the CPU and disks of an RTDBS are scheduled by priority but a standard two-phase locking protocol [Eswa76, Gray79] is used, a high-priority job may be given precedence at the CPU and disks but still get blocked by a lower-priority job because the latter is allowed to hold onto a lock that the high-priority job is waiting for.

The need for all of the components of an RTDBS to be priority-driven requires several changes in the ways that traditional database systems, which are not designed to handle the notion of priority, service their jobs. While existing priority preemptive-resume [Pete86] or priority head-of-the-line [Klei76] scheduling techniques can be used to schedule the CPU in an RTDBS, priority-based algorithms for concurrency control, disk scheduling, admission control, and memory management have to be developed for such a system. Over the past five years, a number of studies have investigated the problems of real-time concurrency control [Abbo88a, Abbo88b, Abbo89, Hari90a, Hari90b, Hari91, Hari92, Hong93, Huan89, Huan91, Kim91] and disk scheduling [Abbo89, Abbo90, Care89, Chen91, Kim91]. However, to the best of our knowledge, no work has met the admission control and memory management challenges that arise in processing queries with deadlines. These are the challenges addressed in this dissertation.

1.1. Real-Time Database System Architecture

Jobs in a real-time application may have either *hard deadlines*, *firm deadlines*, or *soft deadlines*, depending on the extent to which they can tolerate violations of their time constraints. For hard-deadline applications, missing a deadline may have catastrophic consequences, and hence it is necessary to guarantee that all jobs meet their deadlines [Jens85]. Applications like flight control systems and missile guidance systems belong to this category. In contrast, missing a firm deadline or a soft deadline may involve a performance penalty, but it does not have disastrous effects. The distinction between firm- and soft-deadline applications lies in the way that late jobs are treated. For firm-deadline applications, jobs that have missed their deadlines are of no value, and should therefore be aborted and discarded [Hari90a]. In the case of soft-deadline workloads, however, there is some diminished value to completing a job even after its deadline has expired [Abbo88b]. Financial and manufacturing applications usually have either firm or soft deadlines. In this thesis, we will focus on providing database support for firm deadline applications. However, many of the techniques developed here could be applied to soft RTDBSs as well since both firm and soft RTDBSs rely on the same underlying mechanism, i.e. priority scheduling, to help jobs meet their deadlines.

The overall architecture of a centralized firm RTDBS is depicted in Figure 1.1. The system's workload consists of one or more job classes. Each job can either be a query or a transaction. The RTDBS is expected to complete as many jobs on time as possible, discarding those that miss their deadlines. Since certain applications may require any missed deadlines to be distributed among all the classes in a controlled fashion, the RTDBS must also include mechanisms to regulate the relative performance of the individual classes.

The *Priority Mapper* assigns a priority value to every job that is submitted to the system. This value will determine the precedence of the job, relative to other submitted jobs, in receiving resources. In general, the priority of a job depends on its deadline, and may also be contingent on the class that the job belongs to. Furthermore, the priority of a job may remain fixed throughout its lifetime, or the priority may be altered dynamically. Finally, the priority assignment policy may be adjusted over time as a result of feedback on the performance of the system. This priority architecture separates *priority assignment* from *priority usage*, shielding the internal database mechanisms from the details of the priority assignment process so as to minim-

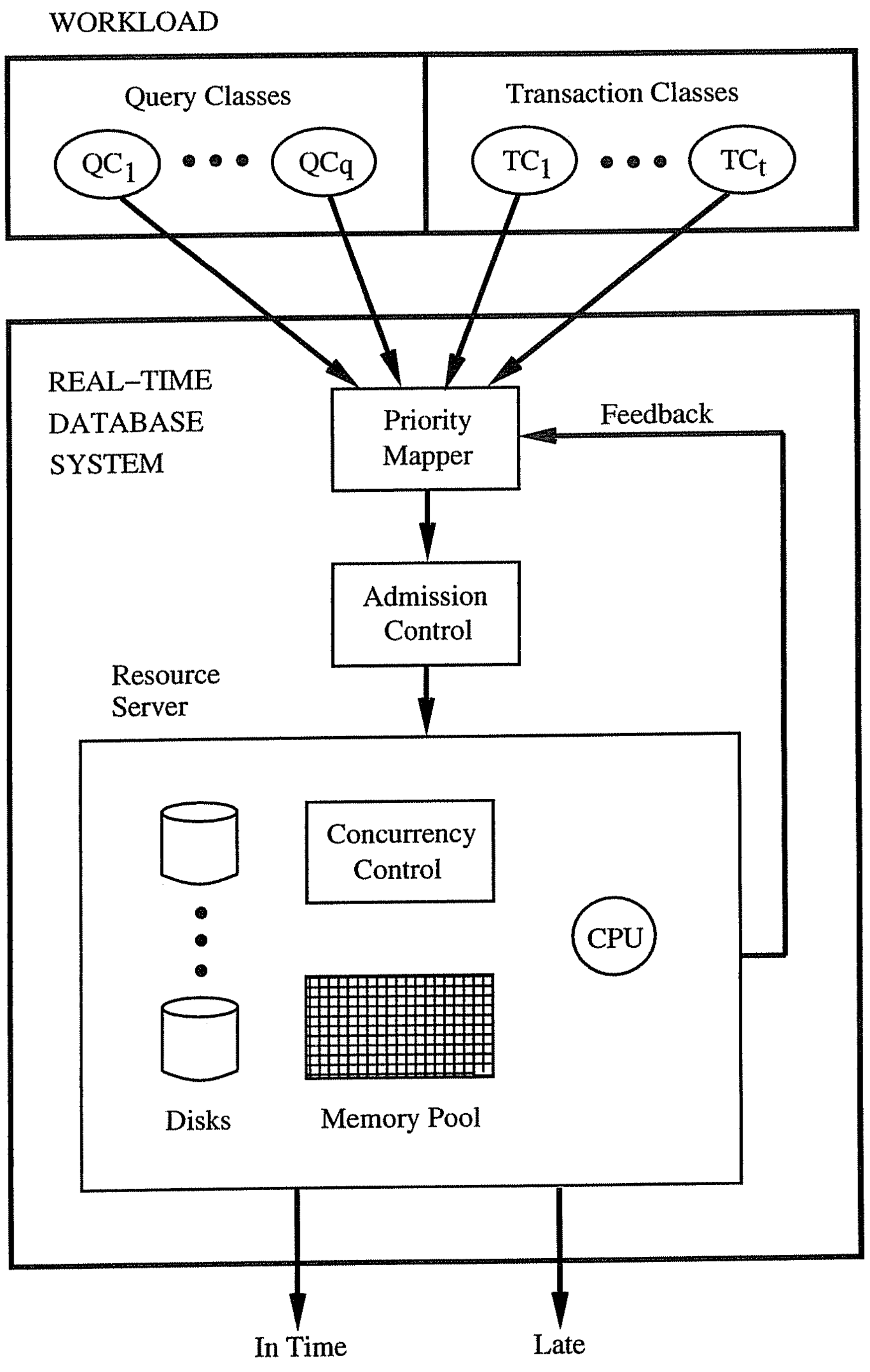

Figure 1.1: Firm Real-Time Database System Architecture

ize the changes that are needed to adapt the database system to a real-time context.

The database system regulates usage of its multiple resources through an admission control mechanism and through its various resource schedulers. The admission control mechanism governs the number of jobs that are allowed to compete for resources at any given time by deciding when (and which) jobs should be admitted. The resources of the database system comprise disks, a CPU, and a memory pool that are shared by the admitted jobs. In making their decisions, both the admission control mechanism and the various resource schedulers take the priority of admitted jobs into consideration, hence making the database system *priority-driven*.

1.2. Thesis Contributions

The RTDBS performance objective of minimizing the number of missed deadlines can be very demanding. This is particularly so in *firm* RTDBSs, where a job loses all value once its deadline expires. In order to accomplish their objective, RTDBSs must employ multiprogramming so that all of their resources can be concurrently utilized to service incoming jobs. Any resource contention that stems from multiprogramming is resolved by the use of priority scheduling. Executing multiple real-time queries that require large amounts of computational memory (e.g., hash tables for joins or tournament trees for external sorts) in such an environment involves admission control and memory management issues that have yet to be addressed. There are two inter-related aspects to the memory management issue: memory allocation, which deals with the way that the RTDBS divides its memory among the admitted queries, and memory usage, which concerns how each individual query takes advantage of its memory allotment.

An RTDBS that is priority-scheduled allots memory to admitted queries according to their relative priorities. This necessitates the RTDBS to re-evaluate, and possibly revise, its memory allocation decisions when a query enters or leaves the system. In order to consistently make the best possible use of its assigned memory, an executing query needs to have the ability to adjust to any revisions in its memory allocation. The research reported in this thesis is conducted in two phases. In the first phase, we investigate memory usage strategies that allow queries to execute efficiently in the face of fluctuations in their memory allocations. This investigation leads us to introduce memory-adaptive processing techniques for hash join and external sorting queries. With these techniques in place, the next step is to equip the RTDBS with admission control and memory allocation policies that exploit the memory-adaptation capabilities of the query processing techniques. The study of these higher-level admission control and memory allocation issues constitutes the second phase of our research. The details of the two phases are presented in the remainder of this section.

1.2.1. Memory-Adaptive Query Processing

A common practice in existing database systems is to allocate a fixed amount of memory to each query (or subquery) throughout its lifetime. Unfortunately, this practice does not work well with priority scheduling due to the fact that certain queries, particularly those that join or sort large relations, can hold on to a large number of buffers for an extended period of time. If such large queries are permitted to hold on to their buffers until they complete, higher-priority queries that arrive after a large query may not be able to execute due to a shortage of memory. This can seriously impede the effectiveness of priority scheduling. Consequently, during the lifetime of a large query, an RTDBS may wish to appropriate some of the query's memory to satisfy the memory requirements of higher-priority queries that arrive; buffers that are taken away may subsequently be returned after those queries leave the system. Given the prospect of continually having memory taken away and given back during its lifetime, it is desirable for a query to be able to continue its execution after losing some of its buffers (and hence be *partially preemptable*) and to subsequently adapt its buffer usage to take advantage of any extra memory that may become available. To simplify our discussion, we shall henceforth refer to these changes in memory allocation as memory fluctuations.

One simple way for an RTDBS to deal with memory fluctuations would be to rely on virtual memory techniques to page the buffers of an affected query into and out of a smaller region of physical memory. If the system detects that this is causing too many page faults, it could suspend the query altogether. An advantage of this approach is that it shields the query processing algorithm from the complexity involved in adapting to memory fluctuations. Unfortunately, paging the buffers of a query is likely to cause buffer pages to be swapped into and out of memory repeatedly, resulting in high I/O overheads when the difference in the amount of available memory and the number of buffers assumed in the query plan is significant. Moreover, suspending queries that are affected by large memory fluctuations reduces the number of active jobs, which may lead to under-utilization of system resources. As we will show later, the performance drawbacks associated with such a paging/suspension approach far outweigh its benefits. In this thesis, we investigate a different approach, namely, to directly involve an affected query in adapting to memory fluctuations. We propose and study the performance of several alternative memory-change adaptation strategies for both hash joins and external sorts, and we show that these adaptive techniques offer effective solutions to the memory fluctuation problem in RTDBSs.

1.2.2. Admission Control and Memory Allocation

A query can execute in a single pass by reading its operand relation(s) and producing its results directly if it is given enough memory. The amount of memory that a query needs in order to complete in one pass is its *maximum* required memory. In memory-constrained situations, many queries can also trade memory for disk I/Os by performing an additional pass. This allows the queries to execute with substantially less memory, but requires the queries to write out temporary files and subsequently read them back in for further processing. The minimum number of buffers that a query needs without having to resort to recursive processing techniques is its *minimum* memory requirement. For instance, a hash join can either execute with its maximum required memory, which is slightly greater than the size of its inner relation, or it can run in an additional pass with as few buffer pages as the square root of its inner relation size [DeWi84, Shap86]. In order to derive the benefits of multiprogramming, it may be necessary for an RTDBS to admit some queries with less than their maximum memory allocations. If too many queries are admitted, however, the resulting additional I/Os could increase the number of missed deadlines, leading to a thrashing condition that makes high concurrency harmful instead of helpful. Multiprogramming is therefore a two-edged sword, and RTDBSs require a priority-cognizant admission control mechanism to protect them against thrashing.

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 of the admitted queries. When the total maximum memory requirement of the admitted queries exceeds the available memory, the RTDBS must decide on the amount of memory to give each query. This decision needs to take into account queries' timing requirements to ensure that queries receive their required resources in time to meet their deadlines. 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 [Corn89, Yu93].

This thesis introduces a priority-cognizant algorithm that dynamically chooses a target multiprogramming level and a memory allocation strategy for queries to balance the demands on the system's memory, CPU, and disks. The algorithm is then augmented with a priority adaptation mechanism to enable it to better handle the demands of multiclass workloads. This extended algorithm strives not only to minimize the number of missed deadlines, but also to distribute the missed deadlines among all classes according to administratively defined workload expectations.

1.3. Organization of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 provides a description of a detailed simulation model of a firm RTDBS which has been used to obtain the performance results reported in this thesis. The methodology employed in using the model for experimentation is also described. In Chapters 3 and 4, we introduce and evaluate memory-adaptive processing techniques for hash join and external sorting, respectively. Chapter 5 presents a query scheduling algorithm that dynamically sets the system's multiprogramming level and memory allocation strategy according to the characteristics of the workload. In Chapter 6, this algorithm is extended to better handle multiclass query workloads. Finally, Chapter 7 summarizes the results of the thesis and outlines avenues for future work.

CHAPTER 2

MODEL AND METHODOLOGY

As described in Chapter 1, the goal of this thesis is the development of memory-adaptive query processing primitives and query scheduling algorithms that are appropriate for use in firm real-time database systems. To aid in evaluating the performance of the various algorithms that will be developed in subsequent chapters, we have implemented a detailed simulation model of a firm real-time database system in the DeNet [Livn90] discrete event simulation language. The simulation model captures a centralized database system, based on the architecture depicted in Figure 1.1. This chapter describes the simulation model and concludes with a discussion of the experimental methodology and performance metrics that underly our experiments.

2.1. Database System Simulation Model

Our simulation model captures the constituents of a centralized real-time database system in seven separate components:

- a *Database* that models the data and its layout;
- 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 buffer replacement policy and the query scheduling algorithm;
- a *CPU Manager* and a *Disk Manager* that are responsible for managing the system's CPU and disks, respectively; and
- a *Concurrency Control Manager* that regulates access to shared data.

Figure 2.1 depicts the overall structure of the database system model and summarizes the key interactions between its components. The details of each component, together with its parameters, are presented below.

2.1.1. Database Module

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 5 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. The parameters of the database module are summarized in Table 2.1.

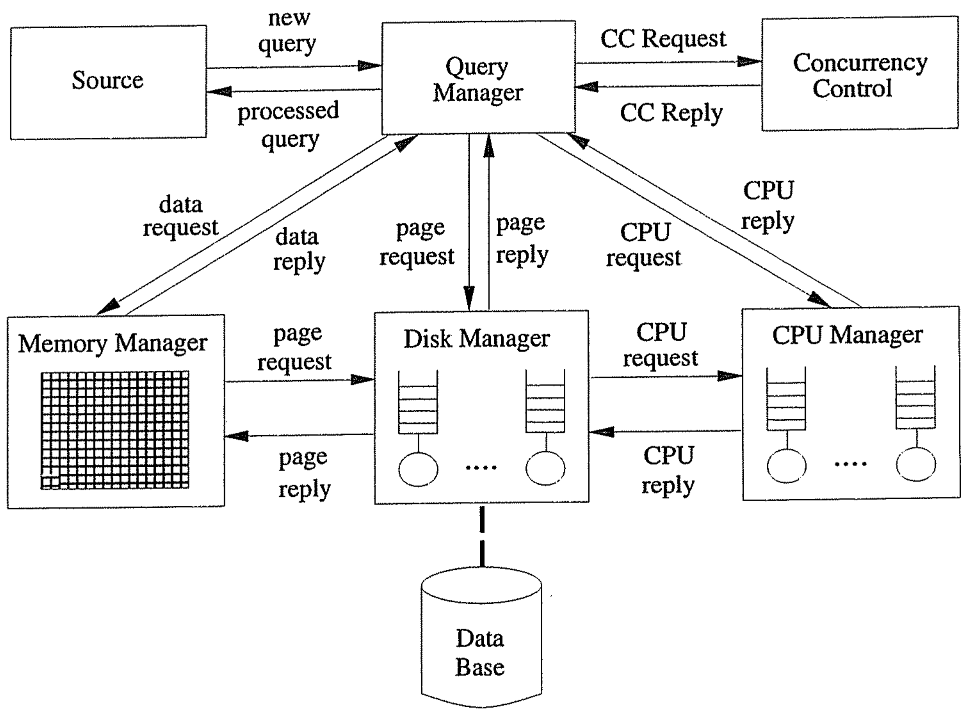

Figure 2.1: Database System Model

DB Model Parameters	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

Table 2.1: Database Model Parameters

2.1.2. Source Module

The Source module represents the real-time application that utilizes the services of the database system. This module is responsible for generating queries for the workload, which are submitted to the Query Manager, and for subsequently receiving the queries after they have been completed or aborted by the Query Manager. The Source also gathers statistics on the returned queries and measures the performance of the system from the application's perspective.

The parameters of the system workload are listed in Table 2.2. It 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 single 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 two relation groups in the set *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. We assume that each tuple in **S** joins with exactly one tuple in **R**, i.e. the join selectivity is $1/|R|$. This is intended to model joins that involve the primary key of one relation and the foreign key of another 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

Workload Parameters	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

Table 2.2: Workload Parameters

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, stand-alone execution time, slack ratio and arrival time of query Q, respectively. The stand-alone execution time of a query is the time it would take to execute alone in the system with its maximum memory allocation, 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. Finally, the parameter F represents the overhead for a hash table. For example, a hash table for a relation **R** is assumed to require $F\|R\|$ pages, where $\|R\|$ is the number of pages in **R**.

2.1.3. Query Manager

The Query Manager component incorporates the Priority Mapper of the RTDBS, assigning a priority to each newly-arrived query according to the underlying priority assignment policy. In addition, this component implements the alternative external sorting and hash join query processing techniques that we will study. In order to expedite the examination of different combinations of external sorting and hash join techniques, we have a single module that is equipped with all of the alternative techniques, as opposed to writing a separate module for each technique. The Query Manager executes each query until it completes or misses its deadline. Queries that complete are labeled "In Time" and returned to the Source module. A query that misses its deadline is immediately "killed"; killing a query involves aborting its execution, marking it as "Late", and then returning it to the Source. While the execution details vary from one query type (hash join versus external sort) to another, and from one technique to a different technique, they all follow the general scheme that is described below.

Upon receiving a query from the Source, the Query Manager first sends a message to the Memory Manager listing the query's minimum and maximum memory requirements and awaits the Memory Manager's admission permission. Once this permission is granted, CPU service is acquired from the CPU Manager to initialize the query. Next, the Query Manager processes the operand relation(s) of the query by carrying out the following sequence of actions for each operand relation page:

- seek access permission from the Concurrency Control Manager;
- ask the Memory Manager to fix the page in memory;
- obtain CPU service[1] to process the tuples in the page;
- allow the Memory Manager to unfix the page;
- issue write requests to the Disk Manager if any output buffers have become full and need to be flushed.

Having processed the operand relation(s), the Query Manager then proceeds to work on any intermediate files that were produced from the relation(s). This involves performing the following steps for each page in the intermediate files:

- ask the Disk Manager to fetch the page;
- obtain CPU service to process the tuples in the page;
- issue more disk requests to write any new output pages.

Finally, the Query Manager sends a commit request to the Concurrency Control Manager, procures CPU service to terminate the query, then returns the query to the Source module. Table 2.3 gives the assumed costs of the various CPU operations involved in the execution of hash joins and external sorts.

Operation	# Instructions	Operation	# Instructions
Common Operations —		**Hash Joins —**	
Start an I/O operation	1000	*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

Table 2.3: Number of CPU Instructions Per Operation

[1] The amount of CPU service required is algorithm-dependent, and could be for the purposes of sorting, hashing, moving, etc.

In order to keep the per-page I/O cost low, if there are buffers that are not already designated for other purposes, a query will spools its output pages and flush them to disk in blocks of *BlockSize* pages when the spool area fills up. Moreover, the Query Manager capitalizes on the prefetch facility of the disks, fetching *BlockSize* consecutive pages on each sequential I/O that incurs a disk cache miss (see Section 2.1.5), except during the merge phase of an external sort. Prefetching is not done for the sorted runs that participate in a merge step because there are likely to be too many runs to make effective use of the disks' limited caches.

Besides handling the data accesses of a query, the Query Manager is also responsible for managing the query's use of memory space. After processing each page of tuples for a query, the Query Manager checks whether there is an outstanding request from the Memory Manager to change the memory allocation of the query. If so, the request is complied with by adjusting the memory space of the query appropriately to free up buffers or to incorporate the additionally allocated memory.

2.1.4. Memory Manager

The Memory Manager is responsible for managing the usage of the system's memory. The memory consists of a pool of M pages which are used for two purposes — to buffer data pages that are frequently accessed, and to serve as the computational memory of admitted queries. Therefore, the Memory Manager embodies the replacement policy for buffer pages and the memory management algorithm that determines admission and computational memory allocation. The details of the buffer replacement procedure are described below; the alternative memory management algorithms will be presented in later chapters.

In the RTDBS, buffers that are not reserved as computational memory are placed in a free list. When a query requires a data page, it first requests that the page be fixed (or pinned) in memory. Upon receiving such a request, the Memory manager checks its free list to see if the required page is there. If so, the buffer that holds the page is fixed to prevent it from being replaced; otherwise a read request is issued to the Disk Manager to fetch the required page into an unfixed buffer from the free list which is then pinned. When the query has finished processing the data page, it informs the Memory Manager so that the buffer can be unfixed and made available for replacement. Since the main concern of this dissertation is the management of queries' computational memory, we implemented a simple LRU replacement policy to manage free buffer pages.

2.1.5. CPU and Disk Managers

The parameters that specify the CPU and disk resources of our model are listed in Table 2.4. A priority head-of-the-line scheduling discipline [Klei76] is used for the CPU. The MIPS rating of the CPU is given by *CPUSpeed*.

Turning to the disk model parameters, *NumDisks* specifies the number of disks attached to the system. Every disk has its own queue that is scheduled by the priority head-of-the-line discipline [Klei76]; any disk requests that share the same priority value are serviced according to the elevator algorithm. Each disk has a 256-KByte cache for use in prefetching pages. The access characteristics of the disks are also given in Table 2.4. Using the parameters in this table, the total time required to complete a disk access is computed as:

$$\text{Disk Access Time} = \text{Seek Time} + \text{Rotational Delay} + \text{Transfer Time}$$

As in [Bitt88], the time required to seek across *n* tracks is computed as:

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

The rotational delay is the spinning time that is needed for the start of the first requested page to be positioned under the disk head. If the current disk access immediately follows an access to a page in the same cylinder, the rotational delay is computed from the number of intervening pages between the current disk head position and the first requested page, using the following formula:

Parameter	Meaning
CPUSpeed	MIPS rating of CPU
NumDisks	Number of disks
SeekFactor	Seek factor of disk
RotationTime	Time for one disk rotation
NumCylinders	Number of cylinders per disk
CylinderSize	Number of pages per cylinder
DiskSurfaces	Number of disk surfaces
PageSize	Number of bytes per page
BlockSize	Number of pages requested on each sequential I/O
M	Total number of buffer pages

Table 2.4: CPU and Disk Model Parameters

$$\text{Rotational Delay} = \frac{\text{Number Of Pages Between Disk Head And Requested Page}}{\text{Number Of Pages Per Track}} \times \text{RotationTime}$$

where the number of pages per track is obtained by dividing *CylinderSize* by *DiskSurfaces*. However, if the current disk access does not immediately follow a disk access to the same cylinder, then the rotational delay is simply set to half of *RotationTime*. Finally, the transfer time is computed as:

$$\text{Transfer Time} = \frac{\text{Number Of Requested Pages}}{\text{Number Of Pages Per Track}} \times \text{RotationTime}$$

2.1.6. Concurrency Control Manager

The Concurrency Control Manager maintains database consistency by regulating access to shared data pages. It services concurrency control requests according to a chosen protocol; these requests can be for read access, write access, committing a job, or aborting a job. We have written separate Concurrency Control Manager modules based on two-phase locking with high-priority conflict resolution [Abbo88b] and optimistic concurrency control with broadcast commit [Mena82, Robi82, Hari90a]. These modules were used in our transaction scheduling studies [Pang92]. For the purposes of this dissertation, which focuses on query scheduling issues, the choice of concurrency control protocol does not matter. Hence, we arbitrarily chose the module that is based on the locking protocol.

2.2. Experimental Methodology and Performance Metrics

As discussed in Chapter 1, there are two major steps in building a real-time database system that handles queries effectively: equipping the RTDBS with memory-adaptive query processing techniques, and providing resource-efficient query scheduling algorithms. This section presents the methodology that we will employ in our examination of alternative query processing techniques and scheduling algorithms, together with the relevant performance metrics.

In Chapters 3 and 4, we develop hash join and external sorting techniques that enable queries to execute efficiently in the face of memory fluctuations. To bring out the performance differences between alternative techniques, we use workloads that are made up of a series of queries; a new query is submitted to the database system only when the previous query has been completed, so that there is only one query in the system at a given time. Random memory fluctuations are provided by a stream of memory requests. These memory requests are deemed to have higher priority than an executing query, so they seize memory from the query and return the memory only when they leave the system. We use the same combination of workloads and memory request streams for each of the alternative techniques, comparing the *Average Query Response Times* that they produce. The hash join and external sorting techniques that consistently yield the shortest average response times are then identified and employed in our subsequent studies on query scheduling algorithms.

The next two research chapters, Chapters 5 and 6, address query scheduling issues, including admission control, memory allocation, and priority assignment, in real-time database systems. There, we use an open queueing model to study the effectiveness of different query scheduling algorithms in managing resource contention among concurrently running queries. Chapter 5, which focuses primarily on single-class workloads, uses the *System Miss Ratio*, defined as

$$\textit{System Miss Ratio} = \frac{\textit{Number of Late Queries}}{\textit{Number of Submitted Queries}}$$

as the primary performance metric. In Chapter 6, where multiclass query scheduling is the subject of our investigation, the query scheduling algorithms are evaluated with respect to the *Class Miss Ratios*, computed as

$$\textit{Class Miss Ratio}_i = \frac{\textit{Number of Late Queries in Class } i}{\textit{Number of Class } i \textit{ Queries}}$$

as well as system-wide measures including the system miss ratio and the *Weighted Miss Ratio*. 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

$$\textit{WeightedMissRatio} = \Sigma\ \textit{Weight}_i \times \textit{Class Miss Ratio}_i$$

where $\textit{Weight}_i$ 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 6.1.

The performance studies in each chapter begin with a baseline experiment. The simulation parameters for this experiment will be selected to force the system to operate in a region where memory contention is the focal point. Subsequent experiments will then be constructed around the baseline experiment, by varying a few parameters at a time, in order to demonstrate the sensitivity of the results of the baseline experiment to these variations. To ensure the statistical validity of our results, we will verify that the 90% confidence intervals for the primary performance metric (average query response times or system miss ratios), computed using the batch means approach [Sarg76], are sufficiently tight. For the results presented, the size of the confidence intervals turn out to be within a few percent of the mean in almost all cases, which is more than sufficient for our purposes. Throughout the thesis we discuss only statistically significant performance differences.

CHAPTER 3

PARTIALLY PREEMPTIBLE HASH JOINS

As described in Chapter 1, queries executing in a priority scheduling environment, such as a real-time database system, face the prospect of continually having memory taken away and given back during their lifetimes. To optimize system performance, query operations that require substantial amounts of memory should be able to adjust gracefully to reductions in their memory allotments; they should also be able to capitalize on any additional memory that becomes available throughout the course of their execution. One important query operation is the hash join, which is the algorithm of choice for processing equi-join queries when the join attributes are not indexed and the query results do not have to be sorted on the join attribute values [Zell90].

To execute efficiently, a hash join requires a significant amount of main memory to hold its hash table and input/output buffers. Depending on the specific algorithm used, the number of buffers that a hash join utilizes ranges anywhere from the square root of the size of its inner relation up to its inner relation size [DeWi84, Shap86], which can be a substantial portion of the system memory. Consequently, a real-time or priority-driven database management system (DBMS) may have to preempt large hash joins in order to satisfy the memory requirements of higher-priority jobs. Unfortunately, preempting a hash join is non-trivial, as most hash join implementations simply allocate a large buffer area when the operation begins and retain the area for the entire duration of the operation. While totally preempting such an implementation by saving and restoring its buffer area is a possibility, doing so is likely to be expensive, particularly if memory fluctuations are common. Thus, other approaches are needed to minimize the performance penalty of preemption.

In this chapter, we investigate alternative strategies for performing hash join operations in a preemptive priority scheduling environment; our aim is to identify efficient strategies for real-time database query processing. Besides studying approaches that deal with memory preemptions by totally suspending affected hash joins or by paging their buffer areas, we also consider algorithms that actively involve the hash joins in adapting to memory fluctuations. These algorithms range from relatively simple ones, which require few extensions to the original hash join algorithm, to more sophisticated algorithms that dynamically adjust the buffer usage of a hash join to reduce the performance penalty resulting from a memory fluctuation. The more sophisticated algorithms form a family of hash join variants that we call *Partially Preemptible Hash Join* (PPHJ) algorithms. The PPHJ variants are all capable of dynamically adjusting their buffer usage in reaction to either a drop (hence the term partially preemptible) or an increase in the amount of memory available for performing the join. They differ from one another in how they prepare for the event of a memory shortage and how they make use of any excess memory. Together, these algorithms cover a wide range of choices in dealing with fluctuations in memory availability.

3.1. Related Work on Hash Joins

In this section, we describe the studies reported in the literature that are related to our work. Before doing so, however, we first introduce some notation that will be used throughout the chapter.

A hash join involves an inner relation **R**, and an outer relation **S**. Relation **R** has $||R||$ pages and $|R|$ tuples. Similarly, relation **S** has $||S||$ pages and $|S|$ tuples. We assume that **S** is the larger relation, i.e. $||R|| \leq ||S||$. We also use a "fudge factor", F, to represent the overhead for a hash table. For example, a hash table for **R** is assumed to require $F||R||$ pages. This notation is summarized in Table 3.1.

Some of the earliest work on joins using hashing is reported in [Kits83]. The GRACE Hash Join algorithm was introduced in that study. In GRACE Hash Join, a join is processed in three phases. First, the inner relation **R** is split into $\sqrt{F||R||}$ disk-resident partitions that are approximately equal in size. In the second phase, the outer relation **S** is partitioned using the same split function. Finally, the **R** and **S** tuples of each

Notation	Meaning
R	Inner relation
S	Outer relation
$\|\|R\|\|$	Number of pages in **R** (similarly for **S**)
$\|R\|$	Number of tuples in **R** (similarly for **S**)
F	Fudge factor

Table 3.1: Notation

disk-resident partition are joined in memory. In the variation of the GRACE algorithm that is presented in [Shap86], a join requires only $\sqrt{F\|R\|}$ output buffers throughout its lifetime. Excess buffers are used to hold subsets of **R** and/or **S** so they need not be written to disk.

A shortcoming of the GRACE Hash Join algorithm is that it does not effectively utilize memory that is in excess of the minimum requirement of $\sqrt{F\|R\|}$ buffers. In [DeWi84], DeWitt et al proposed the Hybrid Hash Join algorithm, which follows the same three phases that GRACE goes through but uses excess memory more effectively. The Hybrid Hash Join algorithm divides the source relations into only as many disk-resident partitions as are necessary to split **R** into subsets that can fit in memory. Each of these partitions is assigned an output buffer. Instead of using the rest of the memory to hold subsets of **R** and/or **S** as in GRACE, this memory is used to hold the hash table for the first partition; the **R** and **S** tuples that belong to this partition can thus be joined in memory directly as **S** is being scanned. The Hybrid Hash Join algorithm was shown to have performance superior to that of GRACE [DeWi84].

The Hybrid Hash Join algorithm is designed to make full use of the memory that a join has available when it first starts execution. During the course of execution, however, there may be a mismatch between the amount of memory that the DBMS can allocate to the join and the size of its **R** partitions. One possible cause of this discrepancy is due to incorrect estimation of the hash attribute distribution. This results in a situation where some **R** partitions are larger than the allocated memory, while other **R** partitions are under-sized. In [Naka88], a modification of Hybrid Hash Join was proposed to deal with this memory misfit problem. Instead of deciding on the number of partitions at the beginning, the proposed modification splits the inner relation into smaller subsets, called buckets, which will later be grouped into partitions. The number of buckets is a parameter of the algorithm. Each bucket is assigned a memory-resident hash table that is initially empty. As **R** is scanned, the buckets gradually grow in size. Each time the memory requirement for the join tries to exceed the available memory, a bucket is written out to disk and all but one of its pages are released. The remaining page is then used as an output buffer for that bucket. After the inner relation **R** has been scanned, there will be as many memory-resident buckets as is possible to fit into the available memory. These buckets are then combined into a single **R** partition that is equivalent to the first partition in Hybrid Hash Join. The disk-resident buckets are also grouped into partitions that will fit snugly in memory when they are brought back in. The next two phases proceed exactly as in the Hybrid (or GRACE) Hash Join algorithm. Through a series of experiments, this modified algorithm was shown to outperform Hybrid Hash Join when the hash attribute distribution cannot be accurately determined [Kits89].

Another factor that can cause a discrepancy between the memory requirement of a join and the memory that is available to it is memory contention due to other transactions or queries (as discussed in the beginning of this chapter), or by other processes that are running in the system concurrently with the DBMS. Zeller and Gray first addressed this situation in [Zell90]. Like the algorithm in [Naka88], the algorithm that they proposed divides the inner relation into many buckets. Unlike the Nakayama et al algorithm, the Zeller and Gray algorithm immediately groups these buckets into tentative partitions. The total number of buckets and the number of buckets per partition are both parameters of the algorithm. Initially, these partitions are each given a memory-resident hash table. As **R** is scanned and the partitions grow in size, the join may attempt to acquire more memory than what is allocated to it. When this happens, a partition will be written out to disk, and the memory that is used for its hash table will be deallocated. This partition now becomes disk-resident, and it is given only an output buffer. Should a partition ultimately turn out to be too big for the allocated memory, the buckets that make up this partition will be regrouped into two smaller partitions. After **R** has been scanned, there will be one or more memory-resident **R** partitions, plus zero or more **R** partitions that reside on disk. Moreover, each **R** partition will be small enough to fit into the allocated memory. The remaining portion of the join proceeds as in phases 2 and 3 of the Hybrid (or GRACE) Hash Join algorithm. The drawback of this algorithm is that when a disk-resident partition gets split (during phase 1), its existing disk pages will contain tuples from the two new partitions. These disk pages will have to be fetched repeatedly during the third phase of the join when disk-resident partitions are processed. The proposed algorithm was prototyped in NonStop SQL, and a preliminary evaluation showed the algorithm to be superior to sort-merge join.

3.2. Memory-Adaptive Hash Join Algorithms

This section gives a detailed description of the memory-adaptive hash join algorithms that we have developed. First, Partially Preemptible Hash Join (PPHJ), a new family of hash join algorithms that dynamically alter the memory usage of joins according to buffer availability, is introduced. We then relate the algorithms proposed in [Naka88] and [Zell90] to PPHJ. Finally, we describe how our implementations of the basic GRACE and Hybrid Hash Join algorithms cope with memory fluctuations.

3.2.1. Partially Preemptible Hash Join

In order to adapt effectively to memory fluctuations, a join has to respond quickly and work with a smaller buffer space when memory is taken away; it must also utilize any additional memory that it is given while executing. These are the main design considerations of PPHJ.

Like the GRACE and Hybrid Hash Join algorithms, PPHJ executes a join in three phases. Phases 1 and 2 partition the inner relation **R** and the outer relation **S**, respectively. During these two phases, the tuples of some **R** partitions are held entirely in memory-resident hash tables, while the tuples of other **R** partitions are stored partly or entirely on disk. To simplify our discussion, we shall henceforth refer to the memory-resident partitions as *expanded* partitions, and the disk-resident partitions as *contracted* partitions. Finally, in the third phase, **S** tuples that reside on disk are fetched and joined with the corresponding **R** tuples. The details of these three phases will become clear shortly.

With PPHJ, the choice of the number of partitions has a significant performance implication. On one hand, we could minimize the number of partitions, as in the Hybrid Hash Join algorithm, by making each contracted partition as large as the initial amount of memory. This would enable the join to make full use of the memory that it starts off with, but would also expose the join to memory fluctuations during phase 3; this is because phase 3 of the join will still require all of the initially allocated memory to build a hash table for each contracted partition. On the other hand, having many small partitions would make the join less vulnerable in phase 3, but would introduce other problems: Since each partition requires at least one page of memory, having more partitions leaves less space in which to expand partitions. To balance the benefit of smaller partitions against the penalty of a larger number of partitions, PPHJ attempts to minimize both the number of partitions and the average partition size. The desired minimum is achieved when the number of partitions is $\sqrt{F\|R\|}$, making the partition size also about $\sqrt{F\|R\|}$. PPHJ therefore divides the source relations into $\sqrt{F\|R\|}$ partitions, the same number of partitions that GRACE Hash Join uses.

Besides rendering joins less vulnerable to fluctuations in memory availability during phase 3, having $\sqrt{F\|R\|}$ partitions rather than the minimum number of partitions has another advantage in that it enables PPHJ to reduce the buffer usage of a join easily during phases 1 and 2 when the need arises. Instead of one big expanded partition, PPHJ maintains several smaller expanded partitions, and each expanded partition has its own hash table. To reduce buffer usage, PPHJ simply contracts one of these partitions by flushing its hash table and freeing all but one page of its memory.

3.2.1.1. PPHJ: The Basics

Having given an overview of PPHJ, we now present the algorithm in detail. The PPHJ algorithm involves five steps. Step (1) initializes the join. Phases 1 and 2 of the join are implemented by steps (2) and (3), respectively. Finally, in phase 3, the join iterates over steps (4) and (5) until all the partitions have been fully processed. Note that the detailed algorithm entails ordering the $\sqrt{F\|R\|}$ partitions. The purpose of this ordering will become clear shortly (once we introduce the variants of PPHJ).

(1) Choose a hash function h and a partition of its hash values that will split **R** into $R_1, .., R_{\sqrt{F\|R\|}}$ and **S** into $S_1, .., S_{\sqrt{F\|R\|}}$, so that each **R** partition will have approximately $\sqrt{F\|R\|}$ pages. An **R** partition can either be "expanded" or "contracted", with the restriction that partition i cannot be contracted before partition i+1. In other words, when needed, we always contract the expanded partition that has the highest index. Each expanded partition requires $\sqrt{F\|R\|}$ pages for its hash table, and each contracted partition needs one output buffer. Expand as many partitions as the allocated memory allows. Any leftover buffers are used as a spool area for pages that are being flushed to disk. The spool area is managed by the LRU policy. In order to reduce disk seeks, spooled pages are flushed out in blocks of several pages each time[1].

[1] In the experiments that are reported in this chapter, spooled pages are flushed out in blocks of 6 pages if the size of the spool is greater than 6; if the spool size is smaller than 6, the entire spool is written out to disk. We selected a block size of 6 pages because, for our system configuration, this choice gives a good compromise between reducing the number of random I/Os, and keeping pages around in the hope that these pages will be fetched again while they are still in memory, thus eliminating some I/O operations. It should be noted that, in [Pang93a], spooled pages are written out one page at a time. This accounts for the different performance figures reported there. However, the relative performance between different algorithms/mechanisms remains the same.

(2) Scan **R**. Hash each tuple with h. If the tuple belongs to an expanded partition, insert the tuple in the hash table of that partition; otherwise the tuple belongs to a contracted partition, so copy it to the corresponding output buffer. In the event that an output buffer becomes full, flush it. After **R** has been completely scanned, flush all output buffers. During this step, memory may be taken away from the join, and this may necessitate contracting more partitions. To contract a partition, flush its hash pages and give away all but one of its allocated pages. The remaining page is then used as an output buffer. When this step is finished, we have a hash table in memory for each expanded partition, and all the contracted partitions are either on disk or in the spool area.

(3) Scan **S**, hashing each tuple with h. If the tuple hashes to a partition of **R** that is currently expanded, probe the corresponding hash table for a match. If there is a match, output the result tuple; otherwise drop the tuple. If the tuple belongs to a contracted partition of **R**, copy the tuple to the corresponding **S** partition's output buffer. When an output buffer fills, it is flushed. After **S** has been completely scanned, flush all output buffers. (Note that additional partitions of **R** can be contracted during this step in response to changes in the amount of memory available to the join.)

Repeat steps (4) and (5) for each partition i that has a nonempty S_i, $i = 1, .., \sqrt{F\|R\|}$. Partition S_i will be nonempty if partition i of **R** was contracted at the start of or at some point during step (3).

(4) If the hash table of R_i is not already in memory, read in R_i and build a hash table for it.

(5) Scan S_i, hashing each tuple and probing the hash table for R_i. If there is a match, output the result tuple, otherwise toss the **S** tuple away. (Note that some pages of R_i and S_i may be in the spool area, thus avoiding I/Os.)

3.2.1.2. PPHJ: Variations on a Theme

When memory is taken away from a join, the basic PPHJ algorithm adapts by contracting partitions; the DBMS suspends the join if fewer than $\sqrt{F\|R\|}$ pages remain. Any additional memory that is given to the join is assigned to a spool area. The following (optional) mechanisms are designed to make more effective use of a join's extra memory.

1. *Contraction.* In step (1) of PPHJ, instead of assigning all $\sqrt{F\|R\|}$ pages to every expanded partition at once, we could let each partition start off with only 1 page, and allocate additional pages to a partition only when the pages that it currently owns are full; all the pages that a partition owns are linked to form a hash chain, as in [Zell90]. This allows all partitions to be "expanded" initially. Under this variation, contraction occurs when an expanded partition requires an additional page and none is available. To distinguish between the original approach of contracting partitions at the start and this variation, we call the former approach *early contraction* and this variation *late contraction.* An advantage of late contraction is that memory may be added after a join has begun execution, thus eliminating the need to contract some partitions.

2. *Expansion.* Throughout step (3), whenever a join has enough free memory to expand the contracted partition that has the lowest index, seize the opportunity and do so. (This is in contrast to just using the additional memory for the spool area.) Expanding a partition involves fetching those of its **R** tuples that have previously been written to disk, so that future **S** tuples that hash to this partition can be joined directly. By arranging to have as many partitions expanded as possible during step (3), this mechanism seeks to minimize the number of **S** pages that ever have to be written to disk.

3. *Prioritized Spooling.* Steps (2) and (3) of PPHJ flush filled output buffers of contracted partitions to disk. These pages can be recalled either in step (3), to re-expand partitions, or in steps (4) and (5), when contracted partitions are processed. Since partitions with lower index numbers are expanded (in step (3)) and scanned (in steps (4) and (5)) before partitions with higher index numbers, we can prioritize the pages in a join's spool area to ensure that pages will be protected from replacement until there is no page belonging to a higher-index partition in the spool area. Moreover, to complement the expansion mechanism, **R** pages are preferred over **S** pages in step (3), so that the spool retains as many **R** pages as possible to facilitate partition expansion. This is expected to improve the effectiveness of spooling as compared to the LRU spooling strategy.

Each of the above mechanisms can be used by itself or can be combined with the other two mechanisms, giving rise to eight PPHJ variants. To differentiate between the variants, we shall postfix a string of the form $X_1X_2X_3$ to PPHJ, where X_1 is either *late* or *early* (late contraction or early contraction), X_2 is either *exp* or *noexp* (expansion or no expansion), and X_3 is either *prio* or *lru* (priority spooling or LRU spooling). Thus, PPHJ(*early,noexp,lru*) denotes the basic PPHJ, which uses early contraction, no expansion and LRU spooling; PPHJ(*late,exp,prio*) denotes the fully enhanced PPHJ, with late contraction, expansion and prioritized spooling, and so on.

3.2.2. Other Algorithms

3.2.2.1. Nakayama et al

The algorithm proposed in [Naka88], which we will call NKT from here on, delays the decision to contract buckets as long as possible. When a bucket has to be contracted, all of its memory-resident pages are flushed to disk without going through the spool area. After contraction, filled output pages of this bucket are spooled if space permits. Therefore, except for its failure to spool pages of contracting buckets, NKT combines late contraction, no expansion, and LRU spooling, using the terminologies of PPHJ. Our context, where the number of buffers allocated to a join may be reduced at any point during its lifetime, necessitates two adaptations to NKT. First, the original NKT algorithm contracts buckets only during phase one of a join. This is inadequate for our purposes, so we allow contractions all through phases 1 and 2. The next adaptation is motivated by the need to keep the size of the **R** partitions as small as possible, so as to minimize the join's vulnerability to memory fluctuations when the **R** partitions are held in memory-resident hash tables. Therefore, instead of grouping several buckets into bigger partitions, we let each bucket form a partition by itself. Finally, the total number of buckets, a parameter of NKT, is set to $\sqrt{F\|R\|}$. This parameter value is chosen to minimize the number of buckets and the average bucket size (as discussed in the beginning of Section 3.2.1), as well as to provide a consistent comparison between NKT and PPHJ. We shall refer to our implementation as NKT' to differentiate it from the original NKT algorithm.

3.2.2.2. Zeller and Gray

Like the Nakayama et al algorithm, the algorithm of Zeller and Gray (which we will refer to as ZG) allows contractions to occur only during the first phase of a join [Zell90]. Our implementation relaxes this restriction so that contractions may occur in both phase 1 and phase 2. The total number of buckets, a parameter of the algorithm, is set to $\sqrt{F\|R\|}$ for the same reason as in NKT'. The number of buckets that make up each partition, another algorithm parameter, is chosen to be one. This choice is motivated by the need to keep the size of the **R** partitions as small as possible, as in the case of NKT'. The resulting algorithm, which we denote as ZG', is equivalent to PPHJ(*late,noexp,lru*).

3.2.2.3. GRACE and Hybrid

Besides PPHJ, NKT' and ZG', we will also include the GRACE and Hybrid Hash Join algorithms in our performance study. Our implementation of GRACE uses $\sqrt{F\|R\|}$ pages for the output buffer of the partitions, and excess buffers are used as an LRU spool area. In the event that less than $\sqrt{F\|R\|}$ pages can be allocated to a join, the DBMS suspends the join altogether. For Hybrid Hash Join, we have implemented two different versions. In the first version, the DBMS suspends a join if it loses any of the buffers that it starts off with; therefore, this version is not partially preemptible. In contrast, the second version resorts to LRU paging whenever the memory available to the join is insufficient to hold its entire hash table. In this case, the join remains executable, so the second hybrid hash join version is partially preemptible. These two versions are denoted by Hybrid(Suspend) and Hybrid(Paging), respectively. With Hybrid(Suspend), all the pages of a join that are written to disk while the join is suspended will be fetched together when the join resumes. This results in sequential I/Os, as opposed to random I/Os which would occur if the disk-resident pages were to be paged in on demand. Hybrid(Paging) does the following for each page that is read in while partitioning/processing relation **S**: Tuples in this **S** page which hash to contracted partitions are copied to the output buffers, while tuples that belong to the (single) expanded partition are joined with tuples in the **R** partition's hash table in two stages. Stage 1 processes those tuples in the current **S** page that hash to pages in the memory-resident portion of the hash table and then discards these processed **S** tuples. **S** tuples that hash to hash table pages that have been paged out to disk are not processed in stage 1. In the second stage of processing an **S** page, all of the disk-resident hash table pages that are required are fetched in order to process the remaining tuples in the current **S** page. During this stage, hash table pages that are replaced are no longer useful to the current **S** page, as the **S** tuples that need these pages of the hash table have already been processed. This two-stage strategy requires knowledge about which hash table pages have been swapped out, and which pages still remain in memory. However, this strategy is superior to a simple strategy that fetches a missing hash table page each time it is demanded by an S tuple, as the simple strategy may repeatedly swap out hash pages that will be used by subsequent S tuples. This would lead surely to unacceptable performance.

3.3. Experiments and Results

In this section, the database system simulator described in Chapter 2 is used to evaluate the performance of the alternative memory-adaptive hash join algorithms. We begin with a baseline model, and further experiments are carried out by varying a few parameters each time. The performance metric of interest here is the average join response time. For ease of reference, the indicator for the algorithms are summarized in Table 3.2. Before we delve into the experiments, however, we first describe how we generate the memory fluctuations.

3.3.1. Source of Memory Fluctuations

To investigate how different join algorithms adapt to fluctuations in the amount of available memory, we simulate an environment where joins have to contend for memory with other jobs that have small memory requirements and, occasionally, with jobs that have large buffer demands. The memory contention experienced by the active joins is modeled here by a simple stream of high-priority memory requests. The duration of the memory requests follows an exponential distribution with a mean of $Duration_{MemReq}$. With a probability of *Prob* (*SmallMemReq*), a memory request takes up a small number of memory pages; otherwise a large

Indicator	Algorithm
PPHJ	Partially Preemptible Hash Join
early versus *late*	Early versus Late Contraction
noexp versus *exp*	No Expansion versus Expansion
lru versus *prio*	LRU versus Priority Spooling
ZG	Zeller and Gray algorithm, same as PPHJ(*late*,*noexp*,*lru*)
NKT′	Nakayama et al algorithm
GRACE	GRACE Hash Join algorithm
Hybrid(Suspend)	Hybrid Hash Join with Suspension
Hybrid(Paging)	Hybrid Hash Join with Paging

Table 3.2: Algorithm Indicators

portion of memory is demanded. The proportion of the total memory that a small request takes up varies uniformly between 0% and *MemReqThreshold*. In the case of a large request, between 0% to 100% of the total memory is taken up.

3.3.2. Baseline Experiment

In our first experiment, we simulate an environment where, except for occasional shortages, there is abundant memory for joins to execute. This environment is simulated by a steady stream of small memory requests and some occasional large memory requests. To achieve this, the mean duration of memory requests is set to 1 second, and *MemReqThreshold* and *Prob(SmallMemReq)* are set to 20% and 0.8, respectively. In other words, 80% of the time a memory request takes up 0-20% of the total memory, and the other 20% of the time the request takes up between 0% and 100% of the total buffer space. A new join is submitted to the system only when the previous join has completed, so that there is only one active hash join at any given time. Moreover, to model primary key-foreign key joins, we let $||R||$ and $||S||$ be 256 pages (2 MBytes) and 2560 pages (20 MBytes), respectively, and **M** be 410 pages (3.2 MBytes). (These parameter values were chosen by scaling the combination $||R||$ = 10 MBytes, $||S||$ = 100 MBytes, and **M** = 16 MBytes down, by a factor of 5, so as to keep the simulation cost down.) Since we are primarily interested in the response times produced by the various algorithms, we set the *SRInterval* parameter to [9999, 9999] so deadline considerations do not enter into the picture here. The resource parameter settings for this experiment are summarized in Table 3.3, while Table 3.4 lists the database and workload parameter settings.

Parameter	**Meaning**	**Setting**
CPUSpeed	MIPS rating of CPU	20 MIPS
NumDisks	Number of disks	1
SeekFactor	Seek factor of disk	0.000617
RotationTime	Time for one disk rotation	16.7 msec
NumCylinders	Number of cylinders per disk	1500
CylinderSize	Number of pages per cylinder	90 pages
DiskSurfaces	Number of disk surfaces	15
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	410 pages

Table 3.3: Resource Parameter Settings for Baseline Experiment

Database	Meaning	Setting
NumGroups	Number of relation groups in the database	2
$RelPerDisk_1$	Number of relations per disk for group *1*	5
$SizeRange_1$	Range of relation sizes for group *1*	[256, 256] pages
$RelPerDisk_2$	Number of relations per disk for group 2	5
$SizeRange_2$	Range of relation sizes for group 2	[2560, 2560] 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	(single)
$SRInterval_1$	Range of slack ratios for class *1* queries	[9999, 9999]
F	Fudge factor for hash joins	1.1
$Duration_{MemReq}$	Mean duration of memory requests	1 second
MemReqThreshold	Max. % buffer demand of a "small" memory request	20%
Prob(SmallMemReq)	Probability of "small" memory request	0.8

Table 3.4: Database and Workload Parameter Settings for Baseline Experiment

Figure 3.1 gives the response time of the various algorithms for this experiment. In the figure, the four PPHJ variants with expansion, i.e. *early,exp,lru*, *early,exp,prio*, *late,exp,lru*, and *late,exp,prio*, deliver the best performance, followed by the two hybrid hash join algorithms. The response time of the remaining four PPHJ variants, i.e. *early,noexp,lru*, *early,noexp,prio*, *late,noexp,lru*, and *late,noexp,prio*, are roughly twice as long as those of the first four PPHJ variants. Finally, the GRACE Hash Join algorithm produces unacceptably long response times — its average response time is more than three times those of the best PPHJ variant. We also collected statistics on the average memory that a join gets upon startup, and found this to be roughly the same for all the algorithms. Hence the behaviors observed here are due to the mechanism(s) of the join algorithms, and not because of a systematic bias in memory allocation. To understand the reason behind these behaviors, we shall analyze each algorithm in turn. In the case of the eight PPHJ variants and ZG', which is equivalent to PPHJ(*late,noexp,lru*), since their response times are determined by three different mechanisms, we shall examine the impact of each of these mechanisms instead. Before doing so, however, we shall first introduce a few terms that will be used to characterize the detailed behavior of the algorithms.

We denote the number of I/Os that a join incurs, excluding those I/Os for reading in the source relations and writing out the results, as "Overhead-I/Os". Overhead-I/Os consist of two components — those associated with **R** partition pages, which we denote as R-I/Os, and those associated with **S** partition pages, which

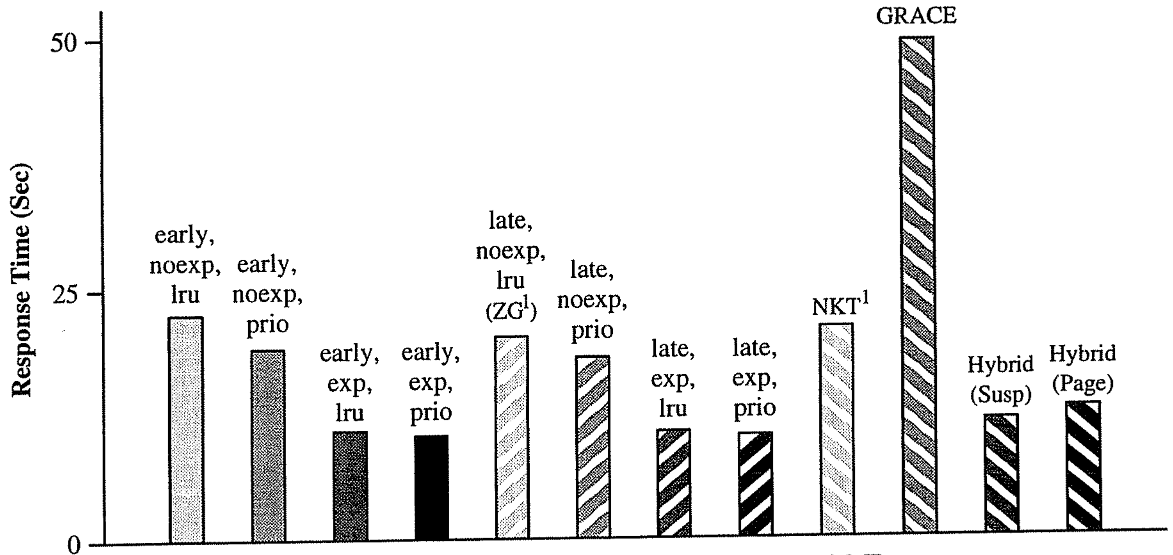

Figure 3.1: $||R||$ = 2 MBytes, $||S||$ = 20 MBytes, **M** = 3.2 MBytes

are denoted as S-I/Os.

Let us first evaluate the expansion mechanism (*noexp* vs. *exp*). Recall that expansion attempts to expand as many partitions as possible during the second phase of a join so as to maximize the number of **S** tuples that are joined directly during this phase. The detailed performance results are listed in Table 3.5, which highlights the performance trade-offs associated with expansion. These results show that expansion is clearly beneficial under the baseline's set of experimental conditions. The reason is as follows: Comparing each set of performance results for no expansion with those for expansion in the same row, we observe that expansion results in slightly more R-I/Os. For example, with late contraction and priority spooling, the last row of Table 3.5 shows that PPHJ requires 275 R-I/Os when there is no expansion and 304 R-I/Os when expansion is activated. This increase is expected because expansion brings in disk-resident pages of **R** partitions during the second phase of a join. These **R** pages may subsequently be swapped out due to another memory shortage, and thus have to be refetched later. Consequently, some **R** partition pages are fetched more than once, resulting in the observed increase in R-I/Os. However, by arranging to expand as many partitions as possible during phase 2 of a join, few **S** pages need to be written out to disk and then processed in phase 3. As an example, refer to the last row of Table 3.5 again. With expansion, the number of S-I/Os is

	No Expansion (*noexp*)				Expansion (*exp*)			
	R-IO	**S-IO**	**Overhead -I/O**	**Resp. Time**	**R-IO**	**S-IO**	**Overhead -I/O**	**Resp. Time**
early,lru	310	1901	2211	22.5	322	77	399	10.9
early,prio	290	1724	2014	19.1	302	77	379	10.4
late,lru	304	1790	2094	20.2	310	72	382	10.7
late,prio	275	1675	1950	18.1	304	75	379	10.3

Table 3.5: Expansion Mechanism

only 75, compared to the 1675 S-I/Os in the case where there is no expansion. This large reduction in S-I/Os more than offsets the drawback in increased R-I/Os, reducing the join response time by more than 40%!

We now examine the priority spooling strategy (*lru* vs. *prio*). To facilitate interpretation of the results, we reorganize Table 3.5 into Table 3.6 to highlight the relative contributions of LRU spooling versus priority spooling. Table 3.6 shows that priority spooling produces some performance improvement over LRU spooling. However, for the two better combinations involving expansion, i.e. *early,exp* and *late,exp*, the performance difference between the two spooling strategies is marginal. The ineffectiveness of priority spooling, when expansion is in effect, is explained as follows: In an environment where there is ample memory and memory shortages are rare, most of the spooled **R** pages are recalled for expansion before they are forced out by occasional memory shortages. Moreover, since expansion keeps most of the **R** tuples in memory-resident hash tables, few **S** tuples need to be written out. The strategy that is used to manage the spool area thus has little impact on performance.

Next, we evaluate the relative merits of early versus late contractions (*early* vs. *late*). Table 3.7 focuses on the impact of the timing of contraction. Late contraction consistently produces lower R-I/Os and S-I/Os

	LRU Spooling (*lru*)				Priority Spooling (*prio*)			
	R-I/O	**S-I/O**	**Overhead -I/O**	**Resp. Time**	**R-I/O**	**S-I/O**	**Overhead -I/O**	**Resp. Time**
early,noexp	310	1901	2211	22.5	290	1724	2014	19.1
early,exp	322	77	399	10.9	302	77	379	10.4
late,noexp	304	1790	2094	20.2	275	1675	1950	18.1
late,exp	310	72	382	10.7	304	75	379	10.3

Table 3.6: LRU versus Priority Spooling

than early contraction, leading late contraction to have lower response times than early contraction. The superior performance of late contraction is explained by the following: By keeping the partitions of a join expanded as long as possible, it may turn out that some partitions need not be contracted after all because additional memory is allocated to the join. Moreover, in the worse case, late contraction will contract only as many partitions as early contraction does. Late contraction thus outperforms early contraction. However, the difference in performance between the two contraction strategies is not substantial, especially when there is expansion. The reason for this is as follows. In phase 1 of a join, early contraction may result in more partitions being contracted than is necessitated by the subsequently available memory. If this happens, however, the excess memory is used to spool the pages of the contracted **R** partitions. Once phase 2 begins, these spooled pages are then recalled to expand partitions so, shortly after the beginning of phase 2, the join is operating with just as many expanded partitions as it would have been with late contraction. As a result, expansion enables early contraction to stay competitive with late contraction.

Turning to NKT' in Figure 3.1, we note that it is similar to PPHJ(*late*,*noexp*,*lru*), except that NKT' writes pages of contracting buckets directly to disk. Thus NKT' loses some of the benefits of spooling if excess memory is not fully utilized. This explains the slightly longer response time of NKT' compared to PPHJ(*late*,*noexp*,*lru*). Clearly, neither PPHJ(*late*,*noexp*,*lru*) nor NKT' is the method of choice for this experiment.

As expected, GRACE Hash Join has the largest response time. Although its small buffer requirement makes GRACE the least vulnerable to memory variability, it fails to exploit the available memory effectively. Instead of joining most of the partitions directly in phases 1 and 2 as in the other algorithms, GRACE simply

	Early Contraction (*early*)				**Late Contraction** (*late*)			
	R-I/O	**S-I/O**	**Overhead -I/O**	**Resp. Time**	**R-IO**	**S-I/O**	**Overhead -I/O**	**Resp. Time**
noexp,lru	310	1901	2211	22.5	304	1790	2094	20.2
noexp,prio	290	1724	2014	19.1	275	1675	1950	18.1
exp,lru	322	77	399	10.9	310	72	382	10.7
exp,prio	302	77	379	10.4	304	75	379	10.3

Table 3.7: Early versus Late Contraction

partitions the source relations during these two phases, and it starts joining the partitions only in phase 3. This approach results in many extra I/Os, of course, which accounts for the relatively poor performance of GRACE.

Finally, we analyze the behavior of Hybrid(Suspend) and Hybrid(Paging). Recall that when a join loses any of the memory that it starts off with, Hybrid(Suspend) allows the DBMS to suspend the join until the lost memory is returned; Hybrid(Paging) pages the hash table of the join within the remaining memory. Since there is ample memory in this experiment, the memory that a join loses is quickly returned. Thus, both versions of the Hybrid Hash Join algorithm perform much better than NKT', ZG' and the PPHJ variants without expansion, as these algorithms contract partitions in response to occasional memory shortages and do not recover from these contractions. However, since a hybrid hash join is not able to utilize extra memory that is allocated during its execution except for spooling, a join that arrives when there is a memory shortage will run with a sub-optimal allocation throughout its lifetime. This is why both Hybrid(Suspend) and Hybrid(Paging) are worse than the PPHJ variants that allow expansion.

To summarize the results of this experiment, we can derive the following conclusions about environments where memory is abundant and the inner and the outer relations differ in size. First, expansion is clearly beneficial, as it produces a considerable reduction in response time by avoiding many I/Os for the larger relation. Second, early contraction and LRU spooling perform only slightly worse than late contraction and priority spooling, respectively, when the expansion mechanism is in effect. Therefore, while Partially Preemptible Hash Join with late contraction, expansion, and priority spooling clearly yields the best performance, all the PPHJ variants with expansion provide feasible alternatives to deal with memory fluctuations.

3.3.3. Memory Contention

In the next experiment, we investigate how the trade-offs between the different algorithms change when we move from an environment where there is ample memory to a situation where memory contention is a severe problem. The total memory size is reduced here to only 40% of $||R||$, while the rest of the parameters are set as in the baseline experiment. Figure 3.2 gives the performance results. We will focus only on behaviors that differ significantly from those observed in the previous experiment.

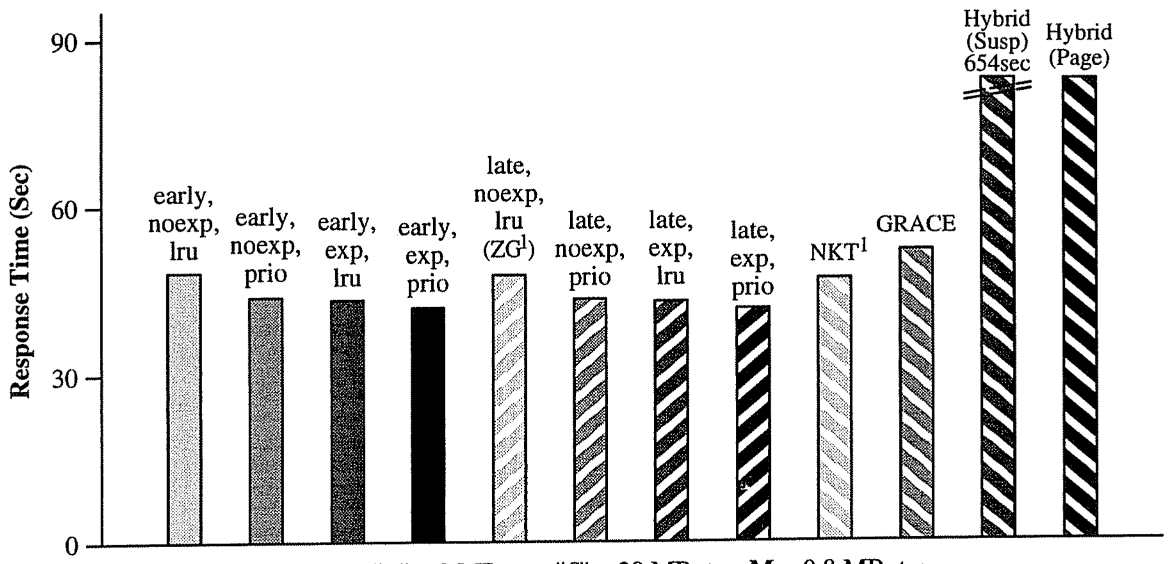

Figure 3.2: ||*R*|| = 2 MBytes, ||*S*|| = 20 MBytes, **M** = 0.8 MBytes

First, we observe that expansion (*noexp* vs. *exp*) now produces only a slight reduction in response time, compared to the 40% performance gain that we obtained in the baseline experiment. To understand this change, we examine the detailed performance results that are presented in Table 3.8. Due to severe memory contention, many of the **R** partition pages that expansion brings in during phase 2 have to be removed when memory availability falls again. These pages will have to be refetched subsequently, which leads to a large increase in R-I/Os with expansion. In fact, expansion roughly doubles the number of R-I/Os. In addition, since the buffer space that is available to expand partitions is limited here, expansion is unable to obtain its previous large increase in the number of **S** tuples that can be directly joined in phase 2. Still, the decrease in S-I/Os more than compensates for the increased R-I/Os.

	No Expansion (*noexp*)				**Expansion** (*exp*)			
	R-I/O	**S-I/O**	**Overhead -I/O**	**Resp. Time**	**R-I/O**	**S-I/O**	**Overhead -I/O**	**Resp. Time**
early,lru	473	4571	5044	48.3	897	3367	4264	43.4
early,prio	472	4570	5042	43.9	816	3360	4176	42.0
late,lru	472	4549	5021	47.8	887	3306	4193	43.1
late,prio	471	4522	4993	43.5	796	3310	4106	41.8

Table 3.8: Expansion Mechanism

Turning our attention to spooling (*lru* vs. *prio*) in Figure 3.2, we again see that priority spooling produces only a slight performance improvement over LRU spooling. In this experiment, where memory shortages occur frequently, few pages are able to remain in the spool area until they are recalled by the joins. This is evident from the large R-I/O and S-I/O values here. For example, with late contraction, no expansion, and priority spooling (*late,noexp,prio*), each join requires an average of 471 R-I/Os. This indicates that about 236 **R** partition pages are written to disk (since each written page involves two I/Os — one to write the page to disk, and another to fetch the page in later for processing); this is more than 90% of the **R** pages. As a result, the spooling policy does not impact performance significantly.

Next, we compare early contraction and late contraction (*early* vs. *late*). As in the previous experiment, late contraction leads to only a small performance gain over early contraction here, but for a different reason. In this experiment, due to the more severe memory contention, few joins are able to retain any large amount of memory for very long. Thus, early contraction and late contraction result in about the same number of expanded partitions, which accounts for their similar response times.

Whereas PPHJ(*late,noexp,lru*) outperformed NKT' in the previous experiment, in this experiment NKT' has a slightly lower response time than PPHJ(*late,noexp,lru*). Since NKT' loses the opportunities to spool pages from contracting partitions, this outcome surprised us initially. A closer examination, however, reveals that this is precisely why NKT' performs better. The reason for this is because, in a memory-constrained situation, most of the spooled pages are eventually written to disk. Instead of writing a few pages out at a time, as in PPHJ(*late,noexp,lru*), NKT' writes out the entire partition that is being contracted, thus resulting in fewer random I/Os than PPHJ(*late,noexp,lru*).

A comparison of GRACE with the other algorithms in Figure 3.2 shows that it is only 20% worse than the best PPHJ variant. Since the main shortcoming of GRACE is its ineffective utilization of excess memory, and the level of memory contention here leaves little excess memory for the active joins, GRACE's conservative use of buffer space yields satisfactory performance. In contrast, Hybrid(Suspend) and Hybrid(Paging) both produce very long response times. In the case of Hybrid(Suspend), joins have long response times because they are often suspended for long periods of time due to memory contention. To understand the poor performance of Hybrid(Paging), consider the following scenario: Suppose an active join just lost some of its memory and, as a result, part of its hash table has been flushed out. The join then fetches the next page of **S** tuples and proceeds to probe the part of the hash table that is in memory. After this, the missing hash table pages have to be fetched in to process this **S** page completely. Before the fetch can be carried out, however, some dirty hash table pages that are currently residing in memory must be paged out to make space for the pages that are about to be fetched in. This at least doubles the number of hash table pages that are written out to disk.

The results of this experiment confirm our previous conclusions that expansion should definitely be attempted when the two source relations differ in size. Moreover, late contraction and priority spooling again produce only slight performance gains over early contraction and LRU spooling.

3.3.4. M to $||R||$ Ratio and $||S||$ to $||R||$ Ratio

The first two experiments lead us to conclude that expanding partitions during the second phase of a join produces a considerable reduction in its response time, and that late contraction and priority spooling lead to some additional savings. We now verify these conclusions by examining the sensitivity of the expansion mechanism to buffer availability and the size of the outer relation. This is achieved by varying **M**, the total number of buffers, while keeping the other parameters constant. The value of those parameters, except for $||S||$ which will be specified later, are those listed in Tables 3.3 and 3.4. For this experiment, we will present only NKT', PPHJ(*late,noexp,lru*)/ZG', PPHJ(*early,exp,lru*), PPHJ(*late,noexp,prio*) and PPHJ(*late,exp,prio*). The other PPHJ variants will not be examined further because their performance was found to be consistently inferior to that of the last three PPHJ algorithms that we have selected to show. GRACE, Hybrid(Suspend) and Hybrid(Paging) are also excluded because they consistently provide unacceptable response times.

In the first part of this experiment, $||S||$ is set to 2 MBytes, the same size as $||R||$. This is intended as a worst case scenario for expansion since a smaller $||S||$ (relative to $||R||$) lowers the number of **S** partition page I/Os that expansion can save. Figure 3.3 plots the response time of the five algorithms against **M**. This figure shows that no algorithm clearly dominate the others in this case. Since the inner relation and the outer relation have the same size, the reduction in S-I/Os that expansion produces just about balances out against the extra R-I/Os that are incurred in expanding partitions, thus explaining the similar response times of PPHJ(*late,exp,prio*) and PPHJ(*late,noexp,prio*). The response times of *NKT*′ and PPHJ(*late,noexp,lru*)/*ZG*′ are almost the same as those of PPHJ(*late,noexp,prio*) in this experiment because, as we have seen in the previous experiments, the choice of LRU versus priority spooling has little influence on performance. Finally, PPHJ(*early,exp,lru*) is comparable to PPHJ(*late,exp,prio*) because there is little difference in performance due to early versus late contraction when expansion is in effect.

For the second part of this experiment, we increase ||*S*|| to 20 MBytes to simulate a condition that is more favorable to expansion (and arguably more typical as well). Figure 3.4 shows the algorithms' response times as a function of **M**. In this case, expansion starts to pay off even for small **M** values. This is because every **R** page that is read in to expand a partition produces, on the average, a ten-fold reduction in S-I/O. Expansion is therefore worthwhile so long as the average number of times that an **R** page has to be refetched due to memory fluctuations is less than the reduction produced for **S**. This is supported by the results for PPHJ(*late,exp,prio*) and PPHJ(*early,exp,lru*), which clearly outperform all of the other algorithms in Figure 3.4. Moreover, the curves for PPHJ(*late,exp,prio*) and PPHJ(*early,exp,lru*) are almost the same, which lends

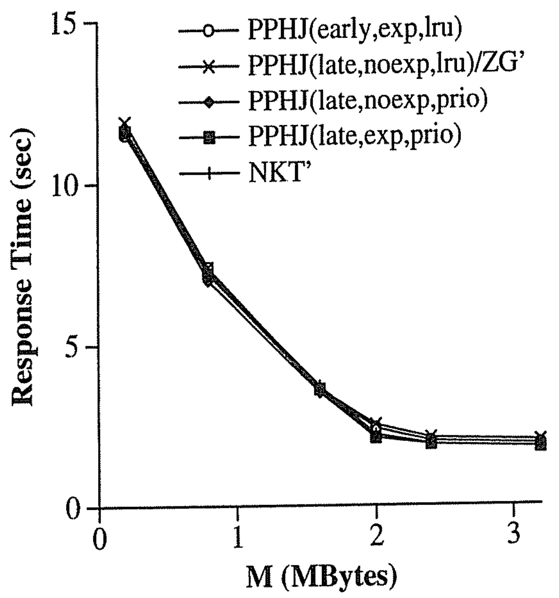

Figure 3.3: ||*R*|| = ||*S*|| = 2 MBytes

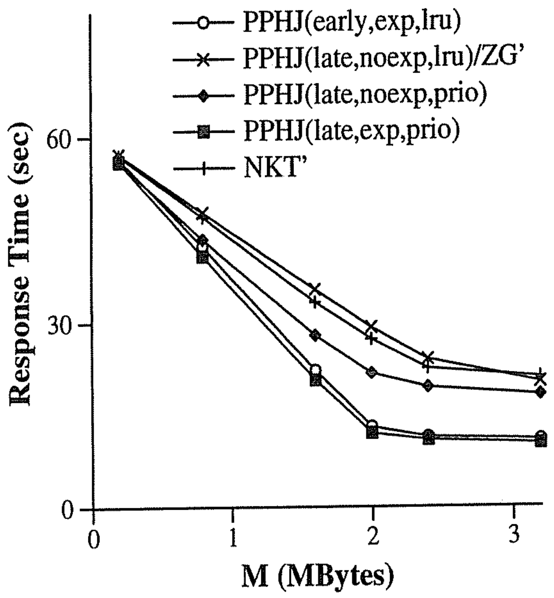

Figure 3.4: ||*R*|| = 2 MBytes, ||*S*|| = 20 MBytes

further support to our previous conclusions that late contraction and priority spooling produce only a slight performance improvement when the expansion mechanism is employed. As for the remaining three algorithms, PPHJ(*late,noexp,prio*) dominates NKT' and PPHJ(*late,noexp,lru*)/ZG' because of the gains from priority spooling, while NKT' is slightly better than PPHJ(*late,noexp,lru*)/ZG' due to the NKT''s use of sequential I/Os.

To summarize, the results of this experiment show that PPHJ with late contraction, expansion, and priority spooling has the best performance over a wide range of **M** to $||R||$ and $||S||$ to $||R||$ ratios. When the $||S||$ to $||R||$ ratio is at its minimum, i.e. $||R|| = ||S||$, PPHJ(*late,exp,prio*) performs as well as any other algorithm that we have examined. As the $||S||$ to $||R||$ ratio increases, the performance difference between PPHJ(*late,exp,prio*) and the other algorithms starts to widen. The only exception to this is PPHJ with early contraction, expansion, and LRU spooling, which emerged as a close second to PPHJ(*late,exp,prio*) here. Therefore expansion should definitely be attempted.

3.3.5. Magnitude of Memory Fluctuations

Our next experiment is designed to explore the sensitivity of the memory-adaptive algorithms to different memory fluctuation magnitudes. Instead of an environment where most of the contenders for system memory are small memory requests, as in previous experiments, here we examine a situation where most of the memory requests are large. We set the parameter *Prob(SmallMemReq)* to 0.2, and keep all the other parameter values as in the previous experiment. In other words, now 80% of the time a memory request takes up 0-100% of the total memory, and the remaining 20% of the time the request takes up between 0% and 20% of the total buffer space. Figure 3.5 gives the resulting response times for $||S||$ = 2 MBytes, while Figure 3.6 presents the performance results for the case where $||S||$ = 20 MBytes.

Turning our attention first to the case where **R** and **S** have the same size, we observe that, as in the previous experiment, no algorithm dominate the others. The most significant difference between Figure 3.5 and the response times obtained in the previous experiment (Figure 3.3) is that here the response time rises more steeply as M goes below 0.8 MBytes. This is because the increased frequency of large memory requests reduces the number of buffers that are available to the hash joins. As **M** is reduced, the average available memory gradually approaches the minimum $\sqrt{F\|R\|}$ buffers that the hash joins require until, at **M** = 0.2 MBytes, the average number of available buffers (15 pages) becomes smaller than $\sqrt{F\|R\|}$ (about 18 buffers). As a result, the hash joins get suspended more frequently, leading to the sharp rise in response time. Another difference between the results obtained from this experiment and the previous experiment is that here the response times are consistently longer, due to the smaller number of available buffers.

Next, we examine the performance results for $\|S\|$ = 20 MBytes, and compare the results in Figure 3.6 with that obtained in the previous experiment (Figure 3.4). Here again the two expansion-based algorithms, PPHJ(*early,exp,lru*) and PPHJ(*late,exp,prio*), are the clear winners. Other than the steeper increase in response time for **M** < 0.8 MBytes and the generally longer response times, the only feature in Figure 3.6 that has not been observed previously is that here the performance gain from priority spooling over LRU spooling is much more pronounced when expansion is not employed, as evident from the curve that corresponds to PPHJ(*late,noexp,prio*) and the curves for PPHJ(*late,noexp,lru*)/*ZG*$'$ and *NKT*$'$. This behavior can be explained as follows: In this experiment, where the increased frequency of large memory requests results in a smaller memory for the hash joins, there is less space to expand partitions. Hence fewer **R** and **S** tuples can

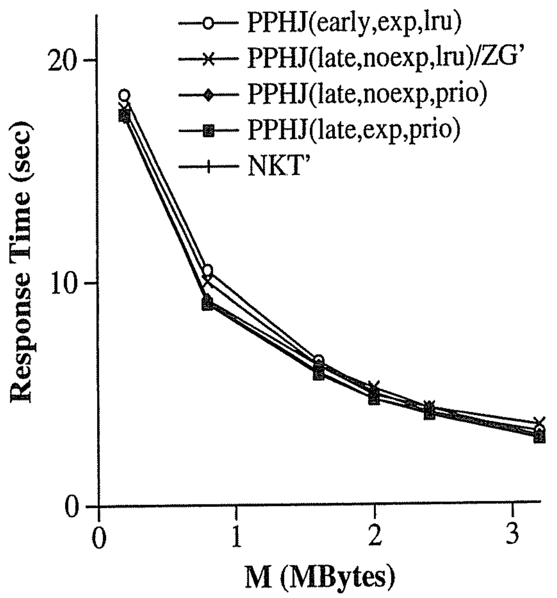

Figure 3.5: $\|R\| = \|S\| = 2$ MBytes

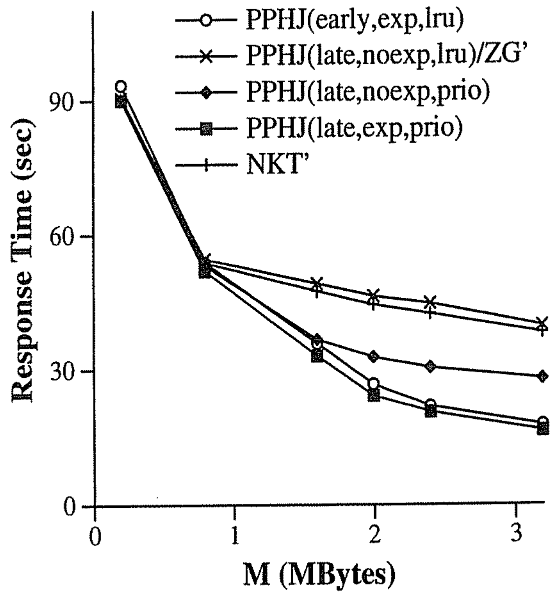

Figure 3.6: $\|R\|$ = 2 MBytes, $\|S\|$ = 20 MBytes

be joined directly in phase 2. This leads to an increase in the number of **R** and **S** pages that have to be written out to disk during phases 1 and 2, and also to a larger number of disk reads during phase 3. The resulting increase in disk I/Os magnifies the performance difference between priority and LRU spooling. Since priority spooling flushes spooled pages by partition, and pages from the same partition reside on contiguous disk pages, priority spooling produces fewer random I/Os than LRU spooling, which accounts for the performance difference observed here. The reason why this difference was not more apparent in Figure 3.5 is because there $||S||$ is only 2 MBytes, so the total number of I/Os is smaller.

The most important conclusion that we can derive from the results of this experiment is that algorithms based on the expansion mechanism are the clear performance winners. Moreover, the expansion mechanism is robust towards different memory fluctuation sizes. Finally, we also noted that when expansion is not employed, priority spooling outperforms LRU spooling in situations where the contenders for system memory are predominantly large memory requests.

3.3.6. Rate of Memory Fluctuations

The expansion mechanism attempts to expand as many partitions as memory permits while the outer relation S is being scanned. In expanding a partition, the DBMS may have to incur some R-I/Os to bring in disk-resident pages of the partition. If the partition remains expanded for a while, the reduction in S-I/Os that result from expanding the partition will gradually offset the cost of expansion. If a memory shortage forces out a partition soon after it is expanded, however, the expansion would not be worthwhile. There is therefore a minimum value for $Duration_{MemReq}$, the average time between consecutive memory fluctuations, in order for expansion to be worthwhile. This section examines the relationship between the cost-effectiveness of expansion and the value of $Duration_{MemReq}$. For the experiments here, **M** is set to 0.8 MBytes to simulate an environment where memory requests have a pronounced effect on the number of buffers that are available for executing joins. Moreover, $Duration_{MemReq}$ is varied between 0.1 second and 10.0 seconds to generate a wide range of memory request interarrival times. The value of the other parameters, except for $||S||$ (discussed later), are those listed in Tables 3.3 and 3.4.

For the first experiment, ||*S*|| is set to 2 MBytes, the same size as ||*R*||. Figure 3.7 presents the response times of the different algorithms. This figure shows that all five algorithms deliver similar performance when $Duration_{MemReq}$ is greater than 1 second, for the same reasons as in previous experiments. When $Duration_{MemReq}$ goes below 1 second, however, expansion has a detrimental effect on system performance, as evident from the curves in Figure 3.7 that correspond to PPHJ(*late,noexp,prio*) and PPHJ(*late,exp,prio*). Hence the minimum $Duration_{MemReq}$ for expansion to be worthwhile is about one second for this experiment.

Next, we increase **S** to 20 MBytes while keeping the other parameters constant. The resulting performance results are given in Figure 3.8. This figure shows that the two expansion-based algorithms, namely PPHJ(*early,exp,lru*) and PPHJ(*late,exp,prio*), outperform the non expansion-based algorithms when $Duration_{MemReq}$ is greater than 0.6 second, whereas the reverse is true when $Duration_{MemReq}$ is less than 0.6 second. This, together with the first experiment, confirm that there is a minimum value for $Duration_{MemReq}$ in order for expansion to be worthwhile. To understand why the minimum $Duration_{MemReq}$ for expansion to be worthwhile occur in the region of 0.5 second to 1 second for both experiments, we shall analyze the detailed I/O costs of partition expansion.

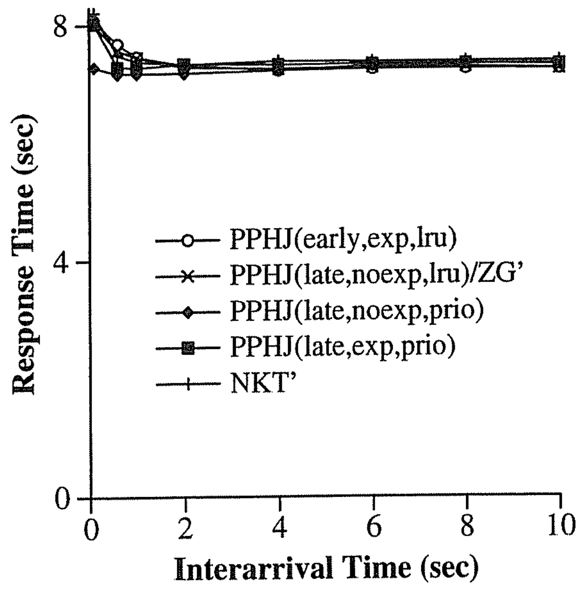

Figure 3.7: ||*R*|| = ||*S*|| = 2 MBytes

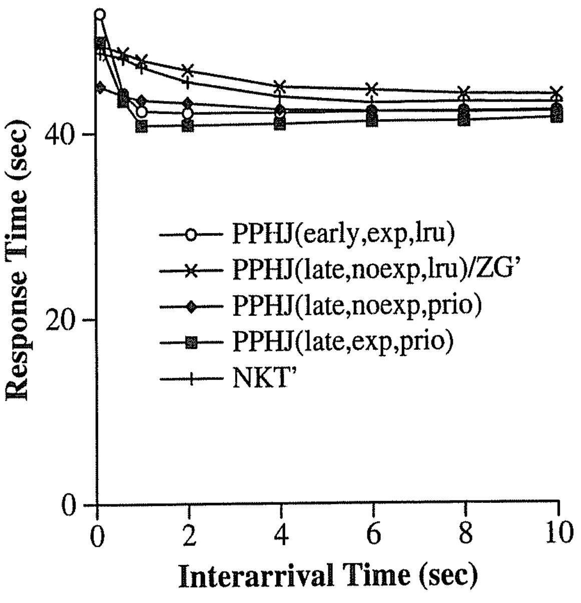

Figure 3.8: ||*R*|| = 2 MBytes, ||*S*|| = 20 MBytes

PPHJ splits each relation into $\sqrt{F\|R\|}$ partitions. Hence each **R** partition has an average of $\sqrt{F\|R\|}$ pages. Denoting the average seek time plus rotational delay by *Seek*, the time to transfer one disk page by *Transfer*, and assuming that the pages of each partition are stored on consecutive disk pages, the cost of expanding one **R** partition is[2]

$$expandCost = Seek + \sqrt{F\|R\|}\ Transfer$$

Suppose a reduction of n **S** pages of this partition is necessary to offset the cost of expansion; n is the quantity we wish to estimate. This reduction results in a saving of n random I/Os, which would otherwise be needed to write out these n **S** partition pages, plus n sequential I/Os to read these pages back from disk when joining the **R** and **S** partitions. Therefore,

$$costReduction = (n + 1)\ Seek + 2n\ Transfer$$

In order for the cost reduction to offset the expansion cost,

$$costReduction \geq expandCost$$

$$\Rightarrow \quad n \geq \frac{\sqrt{F\|R\|}\ Transfer}{Seek + 2\ Transfer}$$

Since each **S** page that is scanned may hash to any of the $\sqrt{F\|R\|}$ partitions, the DBMS needs to scan $n\sqrt{F\|R\|}$ **S** pages to realize a reduction of n pages from expanding one particular partition. We now attempt to quantify the time required to scan $n\sqrt{F\|R\|}$ **S** pages. The cost of reading $n\sqrt{F\|R\|}$ pages is one *Seek* plus $n\sqrt{F\|R\|}$ *Transfer*. Assuming half of the partitions are expanded and the rest are contracted, $n\sqrt{F\|R\|}/2$ of these **S** pages would hash to contracted partitions and have to be written out. Each page that has to be written out while processing **S** incurs one *Seek* to move the disk head from the cylinder that the current **S** page resides on to the cylinder that holds the contracted **S** partition, one *Transfer* to write out the page, and another *Seek* to move the disk head back so that the next **S** page can be read. The time to scan $n\sqrt{F\|R\|}$ **S** pages is therefore

[2] To simplify the analysis, we will ignore the effect of spooling here.

$$scanCost = 1\ Seek + n\sqrt{F\|R\|}\ Transfer + \frac{n}{2}\sqrt{F\|R\|}\ (2\ Seek + 1\ Transfer)$$

$$= (n\sqrt{F\|R\|} + 1)\ Seek + 1.5n\sqrt{F\|R\|}\ Transfer$$

Thus the minimum time needed to expand a partition and then realize enough savings to offset the cost of expansion is

$$minTime = expandCost + scanCost = (n\sqrt{F\|R\|} + 2)\ Seek + \sqrt{F\|R\|}\ (1.5n + 1)\ Transfer$$

Hence, for expansion to pay off, memory shortages should not occur more frequently than *minTime*. Since a memory fluctuation is equally likely to be a decrease or an increase in memory allocation, $Duration_{MemReq}$ has to be at least half of *minTime*.

With our resource parameter settings, reasonable values for *Seek* and *Transfer* are 16 msec and 6 msec, respectively. With an **R** of 2 MBytes and a page size of 8 KBytes, $\sqrt{F\|R\|}$ works out to be 16. Substituting these values into the above equations, *minTime* is about 1.5 seconds. Thus, $Duration_{MemReq}$ has to be at least 0.75 second for expansion to be worthwhile. This explains why, in the above experiments, expansion becomes harmful when $Duration_{MemReq}$ goes below this value.

To summarize, this section demonstrates that expansion is almost always beneficial; the exception is when memory availability fluctuates very rapidly. Given that typical transactions take on the order of a second to complete, and that sorts and joins requiring significant amounts of memory take much longer, it seems unlikely for buffer availability to change so fast as to cause expansion to perform badly in practice. Thus expansion appears to be a generally useful mechanism.

3.3.7. Discussion of Other Alternatives

As described in Section 3.2.2, we have extended the algorithms in [Naka88] and [Zell90] to allow partition contractions during the second phase of a join. An alternative would have been to restrict contractions to only the first phase of a join and, if additional memory is lost during phase two, to suspend the join or to page its hash tables into and out of the remaining memory. We have shown that Hybrid(Suspend) and Hybrid(Paging) both result in long response times, so it is clear that doing suspension or paging with NKT, the Nakayama et al algorithm, and ZG, the Zeller and Gray algorithm, would only worsen their performance.

We therefore did not include those alternatives in our performance study.

In the algorithms studied here, a join is always cognizant of which of its pages are in memory. Another possible approach to dealing with memory fluctuations, as mentioned in Chapter 1, would be to let the DBMS (or the operating system) page the hash table of an affected hash join without informing the join operator. Since a replaced page could be allocated a different memory address space when it is subsequently read in, this approach precludes the possibility of using memory pointers for the hash tables. Instead, logical addresses have to be used, thus resulting in extra overheads for pointer dereferencing. Moreover, using this simple approach, the system could appropriate any of the join's buffers. Since the join operator would have no knowledge of which buffers are paged out, it would access its buffers without attempting to first make use of those buffers that are in memory. This approach would result in even longer response times than Hybrid(Paging), and was therefore not considered. Similarly, the DBMS could simply suspend a join without informing it. This simple approach would be worse than Hybrid(Suspend), which fetches all the pages that have been swapped out when a join resumes execution, as fetching these pages together results in sequential I/Os and lower overheads. This alternative was therefore ruled out too.

3.4. Conclusion

In this chapter, we have addressed the issue of join execution in situations where the amount of memory available to a query may be reduced or increased during its lifetime. These situations will arise in real-time database systems where memory may be appropriated from a join to meet the buffer requests of higher-priority queries, and where additional memory may be made available when other queries complete and free their buffers. In particular, we considered the specific problem of scheduling hash joins, which require large numbers of buffers to execute efficiently and are thus especially susceptible to fluctuations in memory availability. Our study demonstrated that simple approaches that react to a reduction in a join's allocated memory by suspending the join altogether or by paging the hash table of the join into and out of the remaining memory will not produce acceptable performance. There is therefore a need for more sophisticated approaches that enable the join to adapt itself to these memory fluctuations.

To investigate the effectiveness of adapting the buffer usage of hash joins to memory fluctuations, we proposed a family of memory-adaptive hash join algorithms, called *Partially Preemptible Hash Join* (PPHJ). All the PPHJ algorithms split the source relations of a join into a number of partitions that are initially *expanded*, i.e. held in memory-resident hash tables. When the allocated buffers become insufficient to hold all of the partitions, PPHJ responds by *contracting* one of the expanded partitions, i.e. by flushing its hash table to disk and by deallocating all but one of its buffer pages. The remaining page is used as an output buffer for the contracted partition.

Each of the PPHJ variants utilizes additional memory through a (fixed) combination of three mechanisms: *late contraction*, *expansion*, and *priority spooling*. *Late contraction* keeps the partitions of a join expanded as long as possible, i.e. until the buffer usage of the join actually exceeds the allocated memory. In contrast, *early contraction* starts a join by expanding only as many partitions as it estimates will fit into the available memory; the rest of the partitions are immediately contracted. The advantage of late contraction is that additional buffers may be given while the join is executing, thus avoiding the need to contract some partitions altogether. If memory permits, *expansion* fetches contracted partitions of the inner relation **R** into memory-resident hash tables while the outer relation **S** is being partitioned, thereby increasing the number of **S** tuples that can be joined directly without further I/Os. The last mechanism, *priority spooling*, concerns how excess memory is utilized. PPHJ utilizes excess buffers to spool pages that are being flushed to disk, in the hope that these pages will be fetched again while they are still in memory, thus eliminating some I/O operations. By default, the LRU policy is used to manage this spool area. If *priority spooling* is activated, pages in the spool area are prioritized according to the page access pattern of the join so that pages that are likely to be needed first are kept in the spool area. Each of these three mechanisms can be used independently or in conjunction with the other two mechanisms, thus resulting in eight different PPHJ variants.

To understand the performance trade-offs of different hash join algorithms, we carried out a series of experiments using the detailed DBMS simulation model described in Chapter 2. Through these experiments, we confirmed that hybrid hash join with suspension or paging is not satisfactory. Our experiments also revealed that, with one exception, expansion produces a substantial reduction in the response time of a join over a wide range of memory availability and outer versus inner relation sizes. The exception was when memory availability fluctuates extremely rapidly. Moreover, further savings can be achieved by late contraction and priority spooling, though the savings are not nearly as significant as those due to expansion. These findings are important in two ways. First, previous studies [Naka88, Zell90] have proposed algorithms that rely solely on late contraction. Our study showed that expanding partitions while the outer relation **S** is being scanned leads to more effective utilization of excess memory, and hence to lower response times. Second, PPHJ with early contraction, expansion, and LRU spooling was shown to produce response times that were at most 10% longer than that of the best PPHJ variant. Thus for practical reasons it might be desirable to adopt this alternative; this would avoid complicating further the code for the hash join algorithm by incorporating late contraction and priority spooling. However, we will adopt the best variant, namely PPHJ with late contraction, expansion, and priority spooling, for the remainder of this thesis because of its superior performance.

CHAPTER 4

MEMORY-ADAPTIVE EXTERNAL SORTING

As shown in the previous chapter, suspending a hash join or paging its computational memory area are not effective ways for dealing with memory fluctuations during join operations in a priority scheduling environment. Instead, memory-adaptive techniques were introduced to allow a hash join to adjust gracefully to reductions and increases in the amount of allocated memory during its course of execution. This finding leads us to study the memory fluctuation problem for external sorting, another common query operation that requires a large buffer area, in our attempt to furnish the necessary mechanisms for efficient real-time database query processing.

Sorting is frequently used in database systems to produce ordered query results. It is also the basis of sort-merge join [Blas77], a join algorithm employed by many existing database systems, and it is used in some systems for processing group-by queries. An external sort consists of two steps: In the first step, portions of the relation to be sorted are fetched into memory, sorted, and written out in the form of sorted sub-runs. In the second step, which may involve several sub-steps, these sub-runs are merged into a single sorted result. For a large relation, both the sort step and the merge step can potentially utilize many memory pages, and sorting a large relation may take a long period of time. Consequently, as was the case for hash joins in Chapter 3, it may be necessary for memory to come and go throughout the lifetime of a large external sort operation in order for a real-time database management system to satisfy the memory requirements of higher-priority jobs that arrive and leave during the execution of the sort.

In this chapter, we focus on the problem of adapting external sorts to fluctuations in their memory allocations. We propose and study the performance of a memory-change adaptation strategy called *dynamic splitting*. Dynamic splitting adjusts the buffer usage of external sorts, both to reduce the performance penalty resulting from memory shortages and to take advantage of any excess memory that becomes available. In addition, we study how dynamic splitting works with several different in-memory sorting and merging strategies that external sorts can employ. We also show how these techniques can be extended to handle sort-merge joins.

4.1. Standard External Sort Algorithms

An external sort involves two distinct phases. The first phase is a *split* phase, which employs an in-memory sorting method to divide the source relation into a number of sorted runs. The second phase, the *merge* phase, consists of one or more merge steps, each of which combines a number of runs into a single sorted run. The merge phase terminates when only one run remains. Within this framework, the choice of the in-memory sorting method for the split phase is independent of the choice of the merging strategy. This section reviews the common sorting methods and merging strategies that are found in the literature.

4.1.1. In-Memory Sorting Methods

Quicksort and *replacement selection* are two in-memory sorting methods that are commonly used in external sorts. An external sort that employs Quicksort first fills the available memory with as many pages of the source relation as will fit at a time, sorts the tuples in the memory-resident pages, and then writes the result out as a sorted run. This process is repeated until the entire source relation has been scanned. With Quicksort, the length of the runs produced is the size of the memory that is allocated for the split phase.

The second sorting method, replacement selection, works as follows: Pages of the source relation are fetched, and the tuples in these pages are copied into an ordered heap data structure. As more pages are fetched, the heap gradually grows in size until it occupies all of the available memory. At this point, a page of tuples is repeatedly removed from the heap and written to the current run so as to make space for the next incoming page of tuples. The tuples that are removed are those that have the smallest key values (assuming the source relation is to be sorted in ascending order) in the heap, subject to the condition that these tuples must have greater key values than the latest tuple written out in the current run. When none of the tuples in the heap satisfy this condition, the current run ends and a new run is started. On the average, the length of the runs produced by replacement selection is twice the memory allocated for the split phase [Knut73], i.e. twice as long as the runs generated with Quicksort. Hence, replacement selection creates only half as many runs as Quicksort. This could significantly shorten the merge phase that follows. A nice discussion of the details involved in implementing replacement selection can be found in [Salz90].

Although using replacement selection instead of Quicksort can shorten the merge phase, replacement selection is not always the preferred choice because it can also lead to a longer split phase [Grae90, DeWi91]. With Quicksort, there is a cycle of reading several pages from the source relation, sorting them, and then writing them to disk. In contrast, replacement selection alternates between reading a page from the source relation and writing a page to the current run. When the source relation and the run reside on the same disk, this results in many more disk seeks than in the case of Quicksort [Grae90]. In order to reduce disk seeks in replacement selection, a third possible in-memory sorting method is to use replacement selection, but to do block writes, i.e. to write several pages (say N) out to the run each time, instead of only one page at a time as in the original replacement selection procedure. A large N will result in fewer disk seeks, but at the same time it will reduce the average length of the runs. In the extreme case where N is equal to the amount of available memory, this replacement selection variant will fill all of the available buffers with relation pages, then write the sorted pages out together. In this case, the benefit of replacement selection is lost, since the length of the runs becomes the number of available buffers. Thus, the value of N has to reflect a compromise between reducing disk head movements and increasing the average length of the sorted runs. Whether the original replacement selection, Quicksort, or replacement selection with block writes is preferable depends not only on the hardware characteristics of the system, but also on memory allocation and the size of the relation to be sorted.

4.1.2. Merging Strategies

The split phase generates a set of n runs which have to be combined into a single sorted run in the merge phase. The merge phase consists of one or more steps; a merge step takes as input a number of sorted runs and combines them into a longer sorted run. Each input run of a merge step requires an input buffer, and an output buffer is needed for the output run. If at least $n + 1$ pages of memory are available for the merge phase, a single step suffices to combine all of the n runs.

When the source relation is large relative to the available memory, the database system may not be able to allocate enough buffers to a sort operator for it to merge all of its runs in a single step. In this case, preliminary merge steps are required to reduce the number of runs before the final merge can be carried out. Every preliminary merge step incurs extra I/O operations to fetch its input runs from disk and to write out its output run, and there is also extra CPU cost associated with each preliminary step. For this reason, it is desirable for every preliminary step to combine as many runs as the available memory allows, so that there will be as few merge steps as possible. A simple strategy, then, is for each step to merge m - 1 runs, where m is the number of available buffers. Figure 4.1(a) illustrates this strategy for the case where $n = 10$ and $m = 8$. The 10 runs are denoted by R_1, .., R_{10}, and $R_{1\text{-}10}$ denotes the run that results from merging R_1 to R_{10}. In this case, the 10 runs are merged in two steps. The first step merges all the tuples in R_1 to R_7 into $R_{1\text{-}7}$. Step two, which merges $R_{1\text{-}7}$, R_8, R_9 and R_{10} into the final result, begins only after the first step is completed. An alternate strategy is to merge just enough runs in the first step so that each of the subsequent steps merges m - 1 runs. Figure 4.1(b) illustrates the second strategy. The first merging strategy is called "naive" merging, and the second strategy is called "optimized" merging [Grae90]. From Figures 4.1(a) and 4.1(b), it should be apparent that "naive" merging is more expensive than "optimized" merging, as the final step has to process all of the tuples in the relation in both strategies. The preliminary steps incur extra cost, and should therefore merge as few runs as possible (without increasing the number of merge steps) to keep the extra cost down. By merging more runs, "naive" merging increases the cost of the preliminary steps unnecessarily. Thus, the

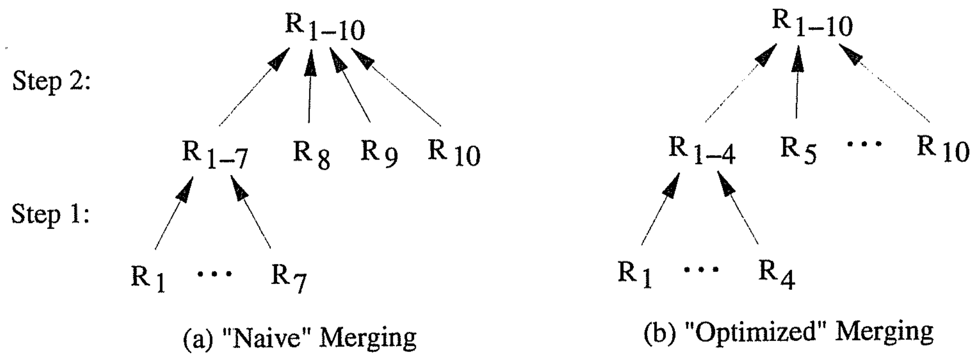

Figure 4.1: Merging Strategies

general rule is to adopt "optimized" merging [Grae91].

Another important aspect of the merging strategy concerns the choice of input runs. All of the merge steps, other than the final merge, have a choice of input runs and should thus merge the shortest possible runs. Such a choice minimizes the cost of the preliminary merges in two ways: Firstly, choosing the shortest runs for a given merge step obviously minimizes its cost. Secondly, the output run of an early merge step may be selected as one of the input runs of a subsequent preliminary merge step. By minimizing the size of the input runs of the early merge step, and hence the size of its output run, the cost of the later merge step is also reduced because it needs to merge fewer tuples. For these reasons, all of the algorithms studied in this paper adopt the policy of merging the shortest possible runs at any given step.

4.2. Memory-Adaptive External Sort Algorithms

In the previous section, we assumed that the amount of available memory remains the same throughout the lifetime of an external sort. As discussed in Chapter 1, however, it is desirable for a sort operator to be able to execute with a varying number of buffers. This section gives a detailed description of a set of alternative memory-adaptive external sort algorithms. Since the in-memory sorting methods for the split phase are independent of the merging strategies for the merge phase, we shall first treat the in-memory sorting methods separately before addressing the merging strategies. Finally, we end the section by introducing some notation that will be used to denote different external sort algorithms throughout the rest of the chapter.

4.2.1. Split-Phase Adaptation

If an external sort is in the split phase when it is asked by the DBMS to release a page, the sort can immediately do so if it has unused buffers, i.e. buffers that are not currently occupied by tuples from the relation. If all of its buffers are in use, however, it will have to clear some of the memory-resident tuples by writing them to output runs, then rearrange the remaining tuples to free up the requested buffers. Next, we consider the case where an external sort is given additional buffers in the split phase. With replacement selection, the new buffer can immediately be used to fill the next incoming page of tuples. In the case of Quicksort, if the external sort is in the process of filling its memory with relation pages, the sort can immediately fill the newly allocated buffers with more relation pages. If the external sort has already started sorting its tuples to create a run, however, the new page will remain unused until the run has been written out and the external sort resumes fetching relation pages.

4.2.2. Merge-Phase Adaptation Strategies

In contrast to the split phase, the merge phase does not adapt to memory fluctuations as easily. One possible solution is to adopt hybrid approaches that allow a sort operator to adapt to memory fluctuations only in the split phase, leaving the DBMS to either suspend an affected external sort or else page its buffers when it is in the merge phase. Besides the drawbacks of suspension and paging that we discussed in Chapter 1, these hybrid approaches would also prevent an external sort from taking advantage of extra memory (beyond the initially allocated amount) that may become available while the sort is in the merge phase. In this thesis, we will therefore explore an alternative, called *dynamic splitting*, that actively involves an affected sort in adapting to memory fluctuations that occur during the merge phase.

4.2.2.1. Suspension

The most straightforward approach to deal with memory shortages that occur during the merge phase of an external sort is for the DBMS to suspend the external sort altogether. The buffers of the external sort can be taken away once it has been suspended. The only information that is needed to resume the merging is the position of the next tuple in each input run. Since the sort operator already keeps track of this information for normal merging operations, no special mechanisms are necessary for suspension. Our implementation of suspension fetches all of the input buffers together when the external sort resumes. This reduces disk seek costs, as opposed to fetching the buffers on demand.

4.2.2.2. Paging

Another obvious way to deal with memory fluctuations during the merge phase is to resort to MRU paging whenever the memory available to an external sort is insufficient to hold all of the input buffers for its current merge step. Our implementation of paging works as follows: The external sort keeps a copy of the current tuple of each input run in its private work space, where the tuples are merged. After writing out the smallest tuple to the output run, the external sort determines which input run this tuple came from, and then attempts to copy the next tuple from this input run. If the buffer for this input run is no longer in memory, the most recently used buffer is selected for replacement, and a disk read is issued to bring the required buffer back in. As with suspension, paging enables an external sort to relinquish its buffers as and when they are needed for replacement or for release to the DBMS.

4.2.2.3. Dynamic Splitting

Dynamic splitting is a strategy that is designed to adapt the merge phase of external sorts to varying memory allocation. When a shortage causes the available memory to go below the memory requirement of an executing merge step, this strategy adapts by splitting the merge step into a number of sub-steps that each fit within the remaining memory. Conversely, when additional buffers are given, the merge steps can be combined into larger steps, i.e. steps that have more input runs, to take advantage of the now-larger memory. The details of the dynamic splitting strategy are presented below.

Suppose that a sort operator is currently executing a merge step, which can be either the final merge of all existing runs or a preliminary merge step. If a memory shortage occurs, causing the available memory to become less than the buffer requirement of the current merge step, the sort operator can immediately stop the current step, split the step into a number of sub-steps, and then start executing the first sub-step. To illustrate this, suppose that the merge phase of an external sort started with 10 runs and 11 buffers, which allowed all runs to be merged at once as in Figure 4.2(a). While the sort is executing this merge step, the available memory is reduced to 8 buffers. The sort operator responds by splitting the merge into a preliminary step that merges R_1 to R_4 into R_{1-4} (assuming "optimized" merging), and a final step that merges R_{1-4} with R_5 to R_{10} into R_{1-10}. After the split, the sort immediately starts to work on the preliminary step. (Note that some of the tuples from R_1 to R_4 have already been merged into R_{1-10} prior to the split, so only the tuples that still remain in R_1 to R_4 will be merged into R_{1-4} by the preliminary step.) This is illustrated in Figure 4.2(b), where the preliminary step, the merge step with the solid arrows, is the one that is being executed. The final step, which has dotted arrows, is inactive. Suppose that no further changes in memory allocation take place, and that the external sort completes the preliminary step without interruption. There are now only 7 runs, and the sort is

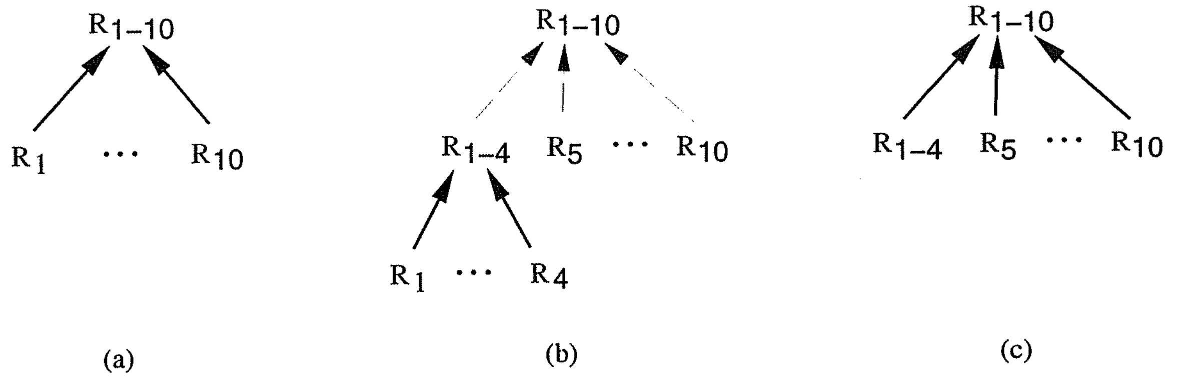

Figure 4.2: Splitting

ready to resume the final merge step. This is achieved by merging $R_{1\text{–}4}$ with whatever is left in R_5 to R_{10}, *appending* the result to $R_{1\text{–}10}$ (Figure 4.2(c)). At this stage, $R_{1\text{–}10}$ contains some of the tuples from R_1 to R_{10} that were merged prior to the split, R_5 to R_{10} each contain some remaining tuples, and the remaining tuples of R_1 to R_4 are now in $R_{1\text{–}4}$. To get the entire sorted result, the sort needs to complete $R_{1\text{–}10}$.

Having discussed how dynamic splitting breaks a merge step into sub-steps in response to a memory reduction, we now present the provision in the dynamic splitting strategy that allows an external sort to combine existing merge steps to take advantage of extra buffers as they become available. We shall introduce this provision by continuing our earlier example. Suppose that, while the sort is executing the preliminary step (the step with the solid arrows) in Figure 4.2(b), the available memory increases to 11 pages again. Instead of completing this step before performing the final merge as discussed previously, the sort operator can switch to the final merge directly. Figure 4.3 illustrates the process involved. At this stage, $R_{1\text{–}10}$ contains some of the tuples from R_1 to R_{10} that were merged prior to the split. To produce the final result, the sort operator needs to append to $R_{1\text{–}10}$ the rest of the tuples that were originally left in R_1 to R_{10}. However, since the sort has already been executing the preliminary step, some of the leftover tuples in R_1 to R_4 are now in $R_{1\text{–}4}$. It is therefore necessary for the external sort to first merge $R_{1\text{–}4}$ with R_5 to R_{10}, appending the result to $R_{1\text{–}10}$. This is shown in Figure 4.3(a), where the final step, which has solid arrows, is now active and the preliminary step is inactive. Once $R_{1\text{–}4}$ becomes empty, the sort operator can proceed to combine the final step with the preliminary step to produce a new final step that again merges the tuples remaining in R_1 to R_{10}, adding them

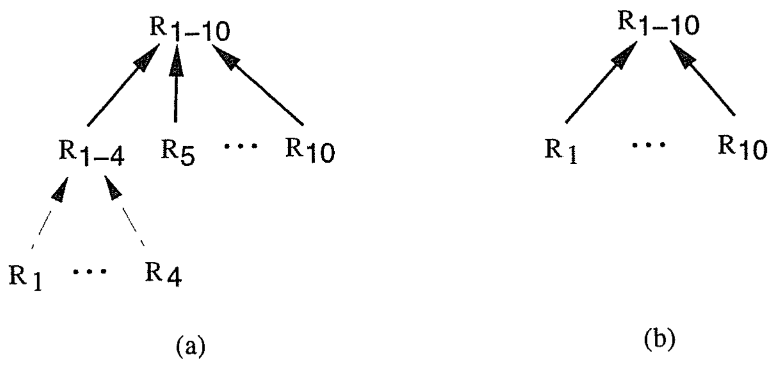

Figure 4.3: Combining Merge Steps

to $R_{1\text{-}10}$ as well (Figure 4.3(b)).

Although our only example shows a split that breaks a merge step into two sub-steps, the splitting procedure can be recursively applied to break a merge step into more than two sub-steps. For example, the preliminary step in Figure 4.2(b) can be split again if memory decreases further while the step is being executed. Similarly, it is possible to combine more than two merge steps by applying the combining procedure recursively. To fully exploit the capabilities of dynamic splitting, the merge phase always starts with a step that combines all of the runs produced in the split phase. If the available memory is insufficient to execute this step, it is immediately split into sub-steps that fit in memory. This enables an external sort to take advantage of excess memory that may become available later by combining existing merge steps into steps that merge more runs, thus helping the sort to recover from a low initial memory allocation if memory happens to be in short supply at the beginning of the merge phase.

There is an important difference between dynamic splitting and the splitting process that was described in Section 4.1.2, which we will call static splitting to distinguish it from dynamic splitting. When an external sort has more runs to merge than its memory allows, static splitting is used to initiate preliminary merge steps to reduce the number of runs. Once started, a merge step has to execute to completion before another merge step can be executed. In contrast to static splitting, dynamic splitting allows an external sort to switch between merge steps, if it so desires, without having to wait for any step to complete. This ability to switch to a different merge step immediately is essential if an external sort is to effectively adapt its buffer usage to both increases and reductions in its allocated memory during the merge phase.

4.2.3. Notation for External Sort Algorithms

In this section, we have discussed three in-memory sorting methods and three merge-phase adaptation strategies. These in-memory sorting methods and merge-phase adaptation strategies will be evaluated in the performance study that follows. In addition, we will compare the relative merits of "naive" merging versus "optimized" merging for the following reason: While "optimized" merging always performs at least as well as "naive" merging *for a fixed memory allocation*, it is not obvious that this is still the case if the amount of memory allocated to a sort operator may be reduced while it is executing. In such situations, "naive" merging may turn out to be better because it utilizes all of the currently available buffers right away (while the sort operator still has them). Since the choice of in-memory sorting method, merging strategy and merge-phase adaptation strategy are all independent, there are 18 possible external sort algorithms, each employing a different combination of in-memory sorting method, merging strategy, and merge-phase adaptation strategy. To differentiate between the algorithms, we shall denote each algorithm by a string of the form $X_1X_2X_3$, where X_1 is either *repl*1, *quick*, or *repl*N (replacement selection, Quicksort, or replacement selection with N-page block reads and writes), X_2 is either *naive* or *opt* ("naive" merging or "optimized" merging), and X_3 is either *susp*, *page*, or *split* (suspension, paging, or dynamic splitting). Thus, for example, *quick,opt,susp* denotes external sort with Quicksort, optimized merging, and suspension. This notation is summarized in Table 4.1.

Parameter	**Meaning**
In-Memory Sorting Method	
*repl*1	Replacement selection
quick	Quicksort
*repl*N	Replacement selection with N-page reads and writes
Merging Strategy	
naive	"Naive" merging
opt	"Optimized" merging
Merge-Phase Adaptation Strategy	
susp	Suspension
page	Paging
split	Dynamic Splitting

Table 4.1: Notation for External Sort Strategies

4.3. Experiments and Results

In this section, the database system simulator described in Chapter 2 is used to evaluate the performance of the alternative memory-adaptive external sort algorithms. We first discuss how the fluctuations in active queries' memory allocations are generated. Next, we begin with an experiment where the amount of memory that is allocated to each external sort remains unchanged throughout its lifetime. This experiment is intended to give us an initial understanding of the trade-offs between different in-memory sorting methods and merging strategies before we delve into the complexities introduced by memory fluctuations. We then present a baseline model that is used to study the performance impact of memory fluctuations, and further experiments are carried out by varying a few parameters each time. The performance metric of interest here is the average sort response time.

4.3.1. Source of Memory Fluctuations

To investigate how different memory-adaptive mechanisms react to fluctuations in the amount of available memory, we again simulate an environment where queries commonly have to contend for memory with other jobs that have small memory requirements and, occasionally, with jobs that have large buffer demands. In contrast to hash joins, which can react to a request to change their memory allocations immediately after processing the tuples in an incoming page, an external sort requires a much longer reaction time if it is in the split phase since it would have to write its memory-resident tuples out to sorted runs and rearrange the remaining tuples before it can free its buffer pages. In order to account for such extended delays, here we explicitly model the high-priority memory requests that compete for memory with the active queries. There are two streams of memory requests, one small and the other large. The generation of small memory requests follows a Poisson distribution with a mean rate of λ_{small}, and the proportion of the total memory that a small request takes up varies uniformly between 0% and *MemReqThreshold*. Moreover, the duration that a small request remains in the system after receiving its required memory is modeled using an exponential distribution with a mean of μ_{small}. Similarly, large memory requests arrive at a mean rate of λ_{large} and have a mean duration of μ_{large}. Each large request occupies between 0% and 100% of the total memory.

4.3.2. No Memory Fluctuation

As mentioned above, our first experiment is designed to study the trade-offs of different in-memory sorting methods and merging strategies in the context of fixed memory allocation. For this experiment, we let $||R||$ be 2560 pages (20 MBytes), and vary **M**, the total system memory. Every external sort will execute with all of the system memory throughout its lifetime. λ_{small} and λ_{large} are both set to 0 request/second, so that there is no memory fluctuation. The CPU and disk model parameters are set according to the values given in Table 3.3. Finally, for the in-memory sorting method *repl*N, we let N be 6 (meaning that tuples are removed from the heap and written out in blocks of 6 pages). This choice was made because, for our system configuration, N = 6 leads to a considerable reduction in the average per-page disk access time over N = 1, as indicated in Table 4.2[1], without incurring the penalty of a significant increase in the number of sorted runs that the split phase generates, as will be evident from our experimental results.

Figure 4.4 presents the response times for the various combinations of in-memory sorting methods and merging strategies. The average number of sorted runs produced by each in-memory sorting method, together with the corresponding average number of merge steps and split-phase duration, are given in Table 4.3. Since there is no memory fluctuation in this experiment, the merge-phase adaptation strategies do not come into play here. The figure shows that all of the response times drop sharply initially as **M** is increased. As **M** grows beyond 0.6 MBytes, however, all of the curves level off. This behavior can be attributed to the reduction in the number of merge steps that takes place as the average number of generated runs decreases. As is evident from Table 4.3, the number of required merge steps initially drops drastically. However, once **M** reaches 0.6 MBytes, all three in-memory sorting methods produce fewer runs than the number of available

N	1	2	4	6	8	10	12
Time	62	36	26	23	22	21	21

Table 4.2: Average Per-Page Disk Access Time (msec)

[1] The average per-page disk access time shown in the table includes the time spent waiting for service, i.e., including waits for completions of previously issued asynchronous disk write requests.

buffers; thus, there can be no further reduction in the number of merge steps (until **M** grows to 20 MBytes, at which point there will be a sudden drop in response time because it will then be possible to sort the entire relation all at once in memory). In this region, increasing **M** leads to fewer sorted runs at the end of the split phase, and hence lower disk seek costs when the runs are merged; this accounts for the slight reductions in response time at the right-hand side of Figure 4.4.

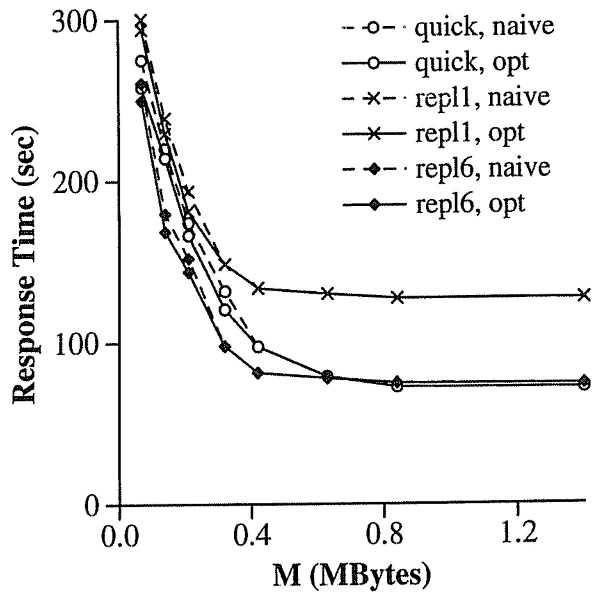

Figure 4.4: No Memory Fluctuation

M MBytes (pages)	0.07 (9)	0.14 (18)	0.20 (27)	0.31 (41)	0.41 (54)	0.61 (81)	0.82 (108)	1.36 (179)
# of Runs								
quick	280	149	101	65	52	34	25	15
repl 1	141	75	52	33	27	18	13	8
repl 6	202	89	57	35	28	19	14	9
# of Merge Steps								
quick	32.0	9.0	4.0	2.0	1.0	1.0	1.0	1.0
repl 1	15.7	4.2	1.9	1.0	1.0	1.0	1.0	1.0
repl 6	22.4	4.9	2.1	1.0	1.0	1.0	1.0	1.0
Split-Phase Duration (sec)								
quick	34	31	29	29	28	27	27	27
repl 1	89	86	85	84	83	83	82	82
repl 6	34	31	31	31	30	30	30	30

Table 4.3: Performance Results for No Memory Fluctuation

Comparing the response times of the three in-memory sorting methods, it is clear that *repl* 1 consistently yields the worst performance. This is due to the large number of random I/Os that *repl* 1 produces, as the external sort alternates between reading a relation page and writing a page to the output run. In contrast, Quicksort writes out an entire run each time, thus producing considerably fewer random I/Os. Quicksort therefore has a much shorter split phase than *repl* 1, which more than offsets the longer merge phase that results from the larger number of runs that Quicksort generates. (Similar observations about the relative trade-offs between Quicksort and *repl* 1 were made in [Grae90] and [DeWi91].) By writing multiple pages instead of only a single page each time as in *repl* 1, *repl* 6 is able to significantly reduce the number of disk seeks in replacement selection, bringing the duration of its split phase much closer to that of *quick*. Moreover, the number of runs that *repl* 6 creates is only marginally more than *repl* 1 in almost all cases. Thus, *repl* 6 is clearly superior to *repl* 1 as a replacement selection procedure. Between *quick* and *repl* 6, *repl* 6 is the winner when $\mathbf{M} < 0.6$ MBytes, whereas *quick* is just slightly faster for $\mathbf{M} > 0.6$ MBytes. The trade-off between *quick* and *repl* 6 is again due to the number of runs that the two approaches generate, relative to the amount of allocated memory. Table 4.3 shows that for $\mathbf{M} < 0.6$ MBytes, *quick* results in more merge steps, and consequently a longer merge phase, than *repl* 6. This is why *repl* 6, which creates significantly fewer runs than *quick*, is superior there. For $\mathbf{M} \geq 0.6$ MBytes, there is enough memory to merge all of the runs produced by *quick* in a single step, so *repl* 6's fewer runs gives it little advantage over *quick*. In this region, the duration of the split phase becomes the dominant factor. Since Quicksort incurs fewer disk seeks than replacement selection, which writes out the pages of each run in several blocks, *quick* is marginally faster than *repl* 6 in this region.

Next, we turn our attention to the two merging strategies, optimized merging (*opt*) and naive merging (*naive*). Figure 4.4 shows that *opt* consistently leads to shorter response times than *naive* for $\mathbf{M} < 0.4$ MBytes, whereas the two merging strategies yield identical performance when $\mathbf{M} > 0.4$ MBytes. Recall that *naive* and *opt* differ in the number of runs that they combine in the first preliminary merge step. The output run of the first preliminary merge step may in turn be combined by a subsequent merge step, the output run of which may be the input of yet another merge step, and so on. The decision of *naive* to include more runs in the first preliminary step thus leads to an increase in the cost of each of these affected steps [Grae91]. The more merge steps there are, the larger the number of affected steps becomes, and consequently the higher the penalty of *naive* gets. For small **M** values, the number of sorted runs that the merge phase has to combine is large relative to the available memory, as shown in Table 4.3. This results in many merge steps, causing the observed differences in response time between *naive* and *opt* in Figure 4.4. Conversely, when **M** is large, the number of merge steps required is small, and so is the penalty of choosing *naive* over *opt*. As **M** increases, the number of merge steps reduces gradually until, when only a single merge step suffices to combine all of the runs, there is no difference between the two merging strategies.

Having now gained initial intuition regarding the performance characteristics and the relative merits of the in-memory sorting methods and merging strategies for fixed memory allocation, we can now proceed to evaluate their performance in the face of memory fluctuations. We will also explore how they interact with the merge-phase adaptation strategies described in Section 4.2.2.

4.3.3. Baseline Experiment

In our baseline experiment, we simulate a situation where the relation to be sorted is much larger than the available memory. This is done by setting $||R||$ to 2560 pages (20 MBytes) and **M** to 41 pages (0.31 MBytes). Small memory requests arrive at an average rate of $\lambda_{small} = 1$ request/second and stay in the system for an average of $\mu_{small} = 0.8$ second. *MemReqThreshold* is set to 20%. Large memory requests arrive at $\lambda_{large} = 0.1$ request/second, and each large request lasts an average of $\mu_{large} = 5$ seconds. The parameter settings for this experiment are summarized in Tables 3.3 and 4.4.

Figure 4.5 gives the response time of the various external sort algorithms for this experiment. The figure shows a wide spread of response times, from a high of 320 seconds produced by *quick,opt,susp* down to a low of 141 seconds, using *repl6,opt,split*. This indicates that the choice of external sort algorithm can have a very significant performance impact. We observe that the four shortest response times are all produced by external sorts that employ *split*. Moreover, the five worse performers all employ *susp*. To understand the reason behind these behaviors, we shall analyze the merge-phase adaptation strategies before considering the in-memory sorting methods and the merging strategies further, as the merge-phase adaptation

Database	Meaning	Setting
NumGroups	Number of relation groups in the database	1
RelPerDisk$_1$	Number of relations per disk for group *1*	10
SizeRange$_1$	Range of relation sizes for group *1*	[2560, 2560] 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	External sort
RelGroup$_1$	Operand relation groups for class *1* queries	{1}
λ_1	Arrival rate of class *1* queries	(single)
SRInterval$_1$	Range of slack ratios for class *1* queries	[9999, 9999]
F	Fudge factor for hash joins	1.1
λ_{small}	Arrival rate of small memory requests	1 request/second
μ_{small}	Duration of small memory requests	0.8 second
MemReqThreshold	Max. % buffer demand of a "small" memory request	20%
λ_{large}	Arrival rate of large memory requests	0.1 request/second
μ_{large}	Duration of large memory requests	5 seconds

Table 4.4: Database and Workload Parameter Settings for Baseline Experiment

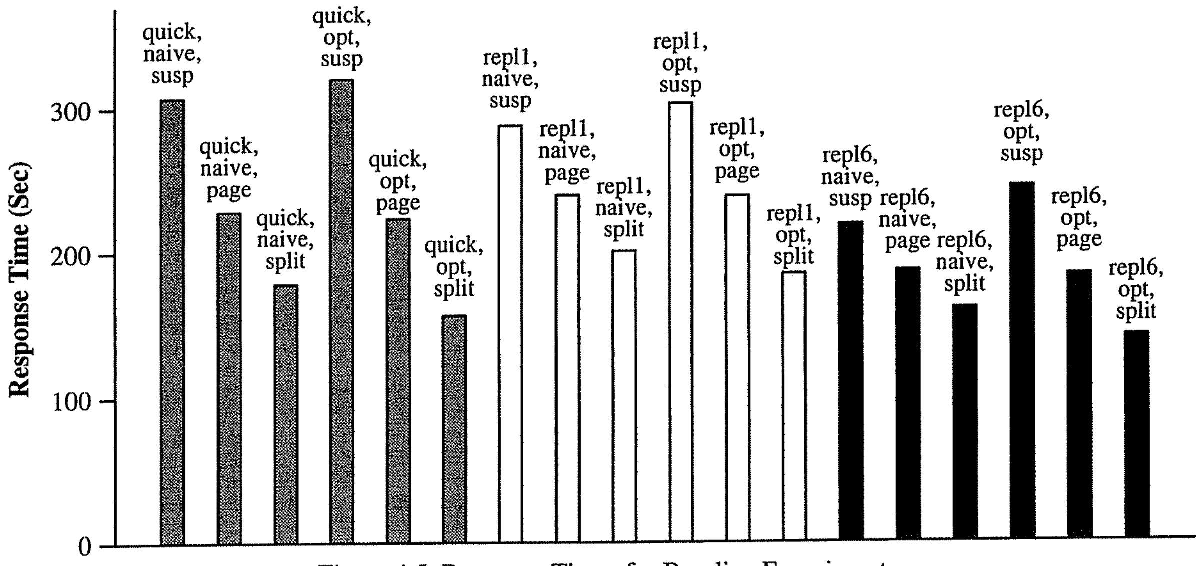

Figure 4.5: Response Times for Baseline Experiment

strategies appear to exert the greatest influence on performance.

The response times given in Figure 4.5 are also listed in Table 4.5, which is organized to highlight the performance trade-offs associated with the different merge-phase adaptation strategies. For example, with Quicksort and naive merging, the first row of Table 4.5 shows that the average response times are, respectively, 307 and 228 seconds when suspension and paging are used, while only 178 seconds are required in the case of dynamic splitting, as indicated by the third column of the same row. This table clearly shows that there is a marked difference between the performance of the three merge-phase adaptation strategies. Among the three, suspension (*susp*) has the worst response times because it does not allow an external sort to make any progress when there is a memory shortage. Paging (*page*) and dynamic splitting (*split*), in contrast, both enable an external sort to keep progressing, which is why they are faster than *susp*. When there is a memory shortage, *page* incurs extra I/Os in paging its input buffers. This is a better alternative than *susp*, but the penalty of paging can be high because the number of extra I/Os is proportional to the extent of the memory shortage. In the case of *split*, an external sort deals with memory shortages by initiating a merge step that fits the remaining memory. This reduces the number of input runs for subsequent merge steps, thereby making them less vulnerable to memory fluctuations. Moreover, *split* is able to take advantage of excess buffers when they become available by switching to a merge step that combines more runs. This is why, as expected, *split* is able to produce shorter response times than *page*.

Next, we evaluate the trade-offs among the in-memory sorting methods. To facilitate interpretation of the results, we reorganize Table 4.5 into Table 4.6 to highlight the impact of the different in-memory sorting methods. Also included in the table are the average duration of the split phase and the average number of sorted runs produced in this phase. In the table, all the algorithms that employ the same in-memory sorting method have the same average number of runs and split-phase duration, as the merging strategies and merge-phase adaptation strategies concern only the merge phase and not the split phase. Due to the much longer split-phase durations that result from excessive disk seeks, as seen in Section 4.3.2, replacement selection (*repl*1) is almost always slower than Quicksort (*quick*) and replacement selection with block writes (*repl*6).

	susp	**page**	**split**
Response Time (sec)			
quick,naive	307	228	178
quick,opt	320	223	156
repl 1,*naive*	287	239	200
repl 1,*opt*	302	238	184
repl 6,*naive*	218	186	160
repl 6,*opt*	244	183	141

Table 4.5: Performance of Merge-Phase Adaptation Strategies

The only exceptions occur when *quick* is used in conjunction with *susp*, which produces the worst response times. The reason is because *quick* generates many more sorted runs than *repl* 1, making the external sorts much more vulnerable to memory shortages. When used with *susp*, the much slower merge phase thus overwhelms any savings that *quick* derives from a shorter split phase. The results also clearly indicate that, as for fixed memory allocations, *repl* 6 outperforms both *repl* 1 and *quick*: *repl* 6 is faster than *repl* 1 due to *repl* 6's much shorter split-phase duration, while *repl* 6 outperforms *quick* because *quick* generates more runs and hence necessitates more merge steps in the merge phase.

We now examine the two alternative merging strategies. Table 4.7 focuses on the relative merits of naive merging (*naive*) versus optimized merging (*opt*). The table shows that *opt* is better than *naive* when used in conjunction with paging or dynamic splitting, while the reverse is true when the merge-phase adaptation strategy is suspension. Recall that *naive* combines more runs in the first merge step, leaving fewer runs to the final merge step. This makes the external sort more vulnerable to memory shortages in the first step than in the final step. In contrast, *opt* attempts to minimize cost by merging as few runs in the first step as possible without increasing the number of merge steps. The result is that the external sort is less vulnerable to memory shortages in the first step, but becomes more vulnerable in the final step due to the larger number of runs that are left until the final step. Since the final step (which has to process all of the tuples in the relation) typically lasts longer than the first step, the net effect is that *opt* makes an external sort more vulnerable to memory shortages than *naive*. Thus, whether *opt* is better than *naive* depends on how much time *opt* saves by merging fewer runs in the first step, as compared to the penalty incurred from exposing the external sort to

	quick	**repl1**	**repl6**
# of Runs	154	99	103
Split-Phase Duration (sec)	28	77	33
Response Time (sec)			
naive,susp	307	287	218
naive,page	228	239	186
naive,split	178	200	160
opt,susp	320	302	244
opt,page	223	238	183
opt,split	156	184	141

Table 4.6: Performance of In-Memory Sorting Methods

memory shortages for a longer period of time. With suspension, an external sort does not make any progress at all when there is a memory shortage, so the penalty of *opt* outweighs its advantage; this explains why *opt* performs badly with *susp*. In contrast to suspension, paging and dynamic splitting enable an external sort to keep progressing during periods of memory shortages. Thus, the penalty of *opt* is not as high, leading *opt* to be beneficial with both paging and dynamic splitting.

To summarize the results of this experiment, we can reach the following conclusions about cases where the relation to be sorted is significantly larger than the available memory. First, dynamic splitting is superior to paging, while suspension results in very large response times and should be avoided. Second, among the three in-memory sorting methods, *repl* 6 combines *repl* 1's advantage of producing long sorted runs and the short-split-phase-duration characteristic of *quick*, making *repl* 6 the in-memory sorting method of choice here. Finally, provided paging or dynamic splitting is used, *opt* is beneficial and preferable to *naive*. Overall, *repl* 6,*opt*,*split* appears to be the most promising algorithm, followed by *repl* 6,*naive*,*split* and *quick*,*opt*,*split*.

4.3.4. M to ||*R*|| Ratio

In the next experiment, we study the sensitivity of the external sort algorithms to different ratios of memory size to relation size. This is achieved by varying **M**, the total number of buffers, while keeping the other parameters constant at their settings of the baseline experiment. In particular, the memory fluctuation rates are the same as in the baseline experiment, and ||*R*|| remains at 20 MBytes so that an increase in **M** results in an increase in the memory to relation size ratio. For this experiment, the in-memory sorting

	naive	**opt**
quick,susp	307	320
quick,page	228	223
quick,split	178	156
repl 1,*susp*	287	302
repl 1,*page*	239	238
repl 1,*split*	200	184
repl 6,*susp*	218	244
repl 6,*page*	186	183
repl 6,*split*	160	141

Table 4.7: Response Time (seconds) for Merging Strategies

methods examined will be limited to Quicksort (*quick*) and replacement selection with block writes (*repl* 6); *repl* 1 will not be considered further because it produces only slightly fewer runs than *repl* 6 while incurring the penalty of a much longer split phase. We will also exclude suspension, since it renders an external sort inactive when memory shortages occur, and is therefore not as effective as paging or dynamic splitting as the baseline experiment showed.

We first examine the performance of the two remaining merge-phase adaptation strategies, dynamic splitting (*split*) and paging (*page*). Figure 4.6 plots the response times for the algorithms that employ replacement selection with block writes (*repl* 6) as a function of **M**. The algorithms that use Quicksort follow the same trends as those in Figure 4.6 and are not shown here. Note that, with the workload parameter settings for this experiment, the range of the average amount of memory available for external sorts here is the same as the range of memory sizes used in Section 4.3.2. Figure 4.6 shows that *split* consistently performs at least as well as *page*: for **M** = 0.1 MBytes, *split* is about 30% faster than *page*, but the difference between their response times narrows considerably when **M** increases; for **M** > 0.6 MBytes, the difference is insignificant. The reason for this trend is that an increase in **M** leads to an increase in the average length of the sorted runs produced in the split phase, producing a corresponding decrease in the number of runs that have to be merged. This makes the external sorts less vulnerable to memory shortages during the merge phase, so there are fewer occasions when paging or dynamic splitting are required. In contrast, a small **M** will increase an external sort's reliance on its merge-phase adaptation strategy, which is why the performance differences between *split* and *page* are more pronounced for smaller **M** values.

We now turn our attention to the in-memory sorting methods, Quicksort (*quick*) and replacement selection with block writes (*repl* 6). The response time of the algorithms based on dynamic splitting, the most promising merge-phase adaptation strategy, are shown in Figure 4.7. The results indicate that *repl* 6 is about 5% faster than *quick* when **M** = 0.1 MBytes. As **M** increases, the response times of the two in-memory sorting methods converge gradually; beyond **M** = 0.9 MBytes, *repl* 6 and *quick* have about the same response times. This trend was also observed in the first experiment (see Section 4.3.2) where external sorts executed with fixed memory allocations throughout their lifetimes. Compared to Figure 4.4 for the first experiment, however, the response time difference between *quick* and *repl* 6 at the left-side side of Figure 4.7 is noticeably

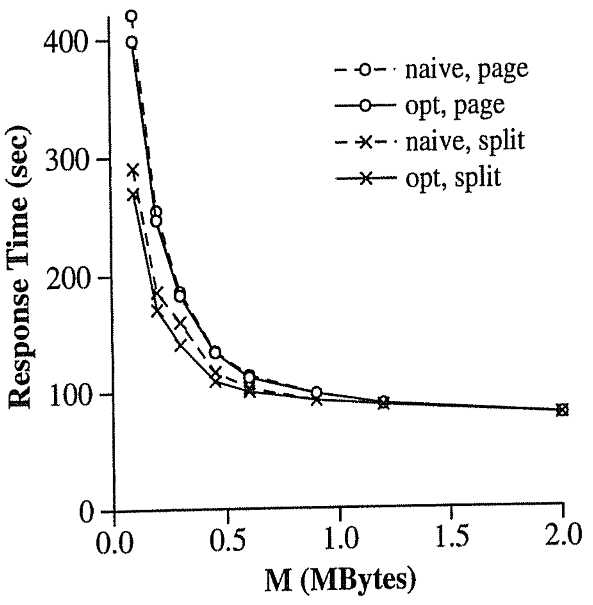

Figure 4.6: *repl*6 (M to ||*R*|| Ratio)

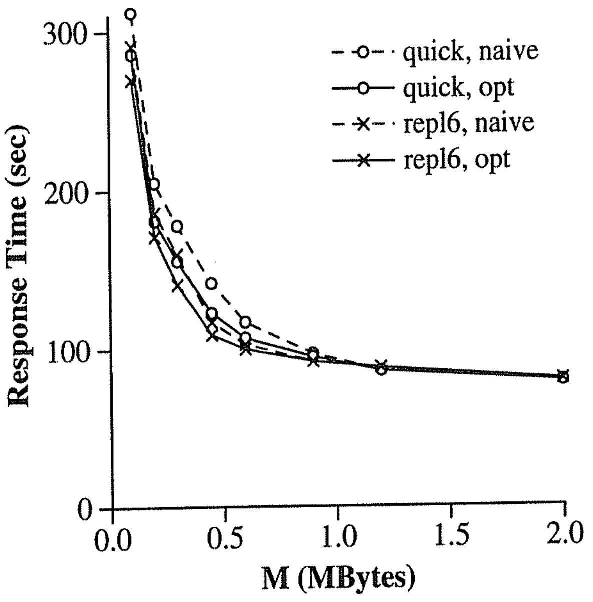

Figure 4.7: *split* (M to ||*R*|| Ratio)

smaller. The reason is because, by sorting and writing out the entire contents of its memory in response to a memory shortage, *quick* frees up all of its buffers so that additional memory requests that arrive while the current run is being generated can be satisfied without requiring further actions on the part of the external sort. *repl*6, in contrast, frees up just enough memory to meet the demands of a waiting memory request. When the next memory request arrives, *repl*6 is forced to write out another block of buffers. Consequently, *repl*6 experiences more interference from competing memory requests than *quick*. This explains *quick*'s performance gains on *repl*6 for M < 0.9 MBytes where external sorts are sensitive to memory fluctuations, though *repl*6 still yields faster response times than *quick* here.

Finally, we evaluate the two alternative merging strategies. Figures 4.6 and 4.7 (shown previously) show the response times for both algorithms that employ naive merging (*naive*) and algorithms that employ optimized merging (*opt*). While these figures cover only a subset of the entire space of eight alternative algorithms, the remaining algorithms give similar results and are not shown. Like the difference between *split* and *page*, there is a significant difference between *naive* and *opt* for small **M** values. When **M** = 0.1 MBytes, *naive* results in a slightly over 5% increase in response time compared to *opt*. The difference between the two merging strategies diminishes steadily as **M** increases until, at **M** = 0.9 MBytes, both strategies yield identical performance. Again, this behavior is similar to what we observed in the static memory allocation case, so we shall not elaborate further on the cause.

The results of this experiment support our baseline experiment's conclusion that, overall, dynamic splitting yields better performance than paging. Moreover, *repl6* leads to shorter response times than *quick*. Finally, among the two merging strategies, optimized merging is the preferred choice.

4.3.5. Magnitude of Memory Fluctuations

Our next experiment is designed to explore the sensitivity of the memory-adaptive mechanisms to different memory fluctuation magnitudes. Instead of an environment where most of the contenders for system memory are small memory requests, as in previous experiments, here we examine a situation where most of the memory requests are large. To achieve this, we interchange the arrival rate and duration of the small and the large requests, so that now $\lambda_{small} = 0.1$ request/second, $\mu_{small} = 5$ seconds, $\lambda_{large} = 1$ request/second, and μ_{large} is 0.8 second. All the other parameters are set as in the previous experiment.

Figure 4.8 highlights the performance difference between dynamic splitting (*split*) and paging (*page*). Compared to the performance results obtained for the previous experiment (shown in Figure 4.6), we note that here both *split* and *page* produce longer response times. Moreover, the difference in response time between *split* and *page* is greater here. These changes are due to the increased frequency of large memory requests, which reduces the number of buffers that are available to the external sorts. This leads to an increase in the number of merge steps in the merge phase, and lengthens the response time of *split*. For example, for **M** = 0.1 MBytes, *split*'s response time is now 345 seconds, whereas it was only 280 seconds previously. In the case of *page*, there are additional factors that adversely affect the performance of the external sorts: When the actual number of buffers that an external sort has is smaller than the buffer requirement of an executing merge step, the penalty in extra I/Os that paging incurs is proportional to the extent of the memory discrepancy. In this experiment, where memory availability fluctuates more widely, the penalty of paging is magnified by the larger memory discrepancies. Moreover, an external sort based on paging is unable to utilize memory that is in excess of its initial memory allocation. This handicap causes paging to suffer from memory fluctuations; moreover, the larger the memory fluctuations, the greater an impact this handicap exerts on sort performance.

Together, these two factors slow down the performance of *page* over and above the performance penalty already imposed by the larger number of merge steps. In particular, they account for the 120-second hike in *page*'s response time, from an average of 410 seconds in Figure 4.6 to an average of 530 seconds here, compared to the smaller 65-second increase in the case of *split*.

The performance results for the two in-memory sorting methods, Quicksort (*quick*) and replacement selection with block writes (*repl*6), are shown in Figure 4.9. From the figure, it is apparent that the increase in the magnitude of memory fluctuations narrows the performance difference between *quick* and *repl*6 as compared to the previous experiment (Figure 4.7). The reason is because here frequent large memory shortages force *repl*6 to write out many memory-resident tuples each time. This hampers *repl*6's ability to keep a large selection of tuples in memory and to write out only those tuples that have small key values, leading to shorter output runs. As a result, the number of runs that *repl*6 produces becomes much closer to that of *quick*.

Finally, we examine how the change in memory fluctuation magnitude impacts the merging strategies. The response times of some algorithms that employ naive merging (*naive*) and others that are based on optimized merging (*opt*) are given in both Figures 4.8 and 4.9. In this experiment, where external sorts frequently

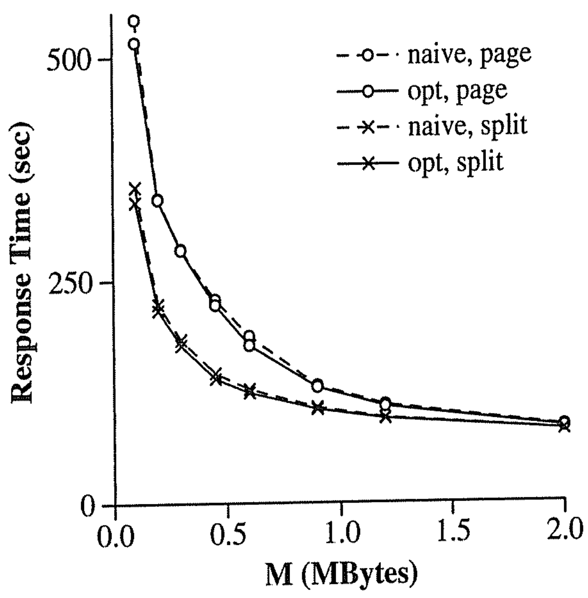

Figure 4.8: *repl*6 (Memory Fluctuation Magnitude)

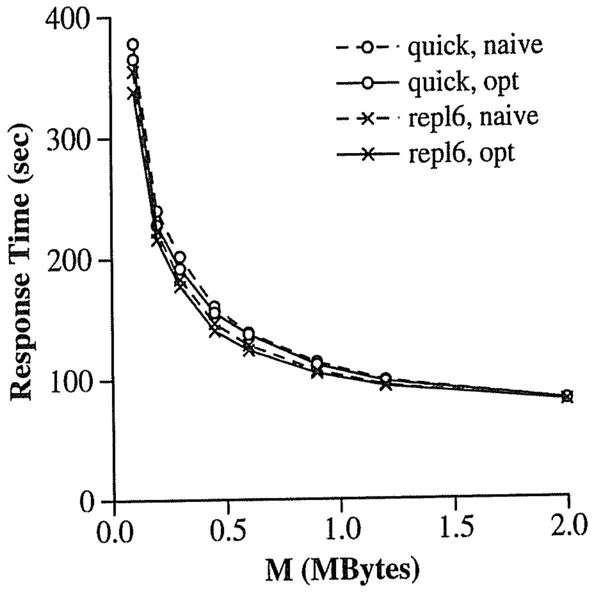

Figure 4.9: *split* (Memory Fluctuation Magnitude)

experience large fluctuations in their allocated memory, the number of runs that an external sort selects for the first preliminary merge step during a split, whether according to *naive* or based on *opt*, often turns out to be sub-optimal because of significant changes that occur in the external sort's memory allocation during the preliminary merge steps. Thus, *opt* is not that much better than *naive* here, in contrast to the previous experiment where memory allocation was less volatile.

In summary, this experiment reveals that large fluctuations in memory availability accentuate the importance of the merge-phase adaptation strategy, while diminishing the differences between the alternative in-memory sorting methods and merging strategies. Again, the results reinforce our previous conclusions about the usefulness of dynamic splitting in dealing with memory fluctuations.

4.3.6. Rate of Memory Fluctuations

Our last experiment for external sorts is designed to investigate how different memory fluctuation rates might affect the relative performance of the merge-phase adaptation strategies and the in-memory sorting methods. We vary the fluctuation rates by first lowering them to λ_{small} = 0.2 request/second and λ_{large} = 0.02 request/second. To ensure that this does not change the average available memory from that in the baseline experiment, the duration of the memory requests are prolonged by the same factor, i.e. μ_{small} = 4 seconds and μ_{large} = 25 seconds. Next, we raise the rate of fluctuation by a factor of 25, setting λ_{small} = 5 requests/second, μ_{small} = 0.16 second, λ_{large} = 0.5 request/second, and μ_{large} = 1 second.

Figures 4.10 and 4.11 show the performance results of four of the external sort algorithms for both the fast and slow memory fluctuations (labeled *fast* and *slow*, respectively, in the figures). In the figures, the solid lines show the response times of the algorithms, while the dotted lines give the split-phase durations. The solid curves in these two figures show that, while the relative performance of the algorithms remains the same as in our previous experiments, the change in memory fluctuation rate does have an impact on the response time of the algorithms for small **M** values. The figures also indicate that when **M** is large, increasing fluctuation rate has little impact on the response time because external sorts are not sensitive to memory fluctuations in this region, as discussed in previous experiments. As **M** is decreased, external sorts become vulnerable to memory fluctuations, and switching the fluctuation rate parameters from their slow settings to their fast settings increases the response times of all four external sort algorithms shown here. For paging, the reason is that, when memory allocation increases after a shortage, paging requires some time before the pages that have been swapped out can be brought back in to fill the newly allocated memory. During this time, the effective number of buffers used is less than the allocated memory. Therefore, when the memory fluctuation rate increases, the effective memory utilization goes down and this leads to longer response times. In the case of dynamic splitting, external sorts react to changes in memory allocation by switching merge steps. Each switch incurs some overhead in bringing the input and output buffers of the new step into memory, so dynamic splitting is also adversely affected by increased memory fluctuations. After **M** = 0.3 MBytes, however, further reduction in **M** narrows the gap between the response times for the slow and the fast fluctuation settings. This phenomenon is due to the fact that, as **M** decreases, so does the magnitude of the memory fluctuations, and hence the performance penalty imposed by these fluctuations. This is why external sorts suffer less from the more frequent fluctuations when the buffer size is very small than when **M** is slightly larger. In contrast to the merge-phase adaptation strategies, the in-memory sorting methods are insensitive to changes in the fluctuation rate, as indicated by the dotted lines in Figures 4.10 and 4.11, since the average available memory remains the same despite changes in the fluctuation rate.

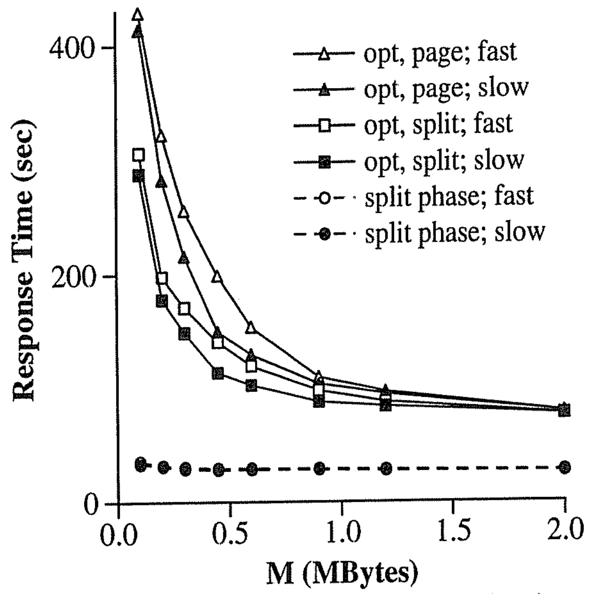

Figure 4.10: *quick* (Memory Fluctuation Rate)

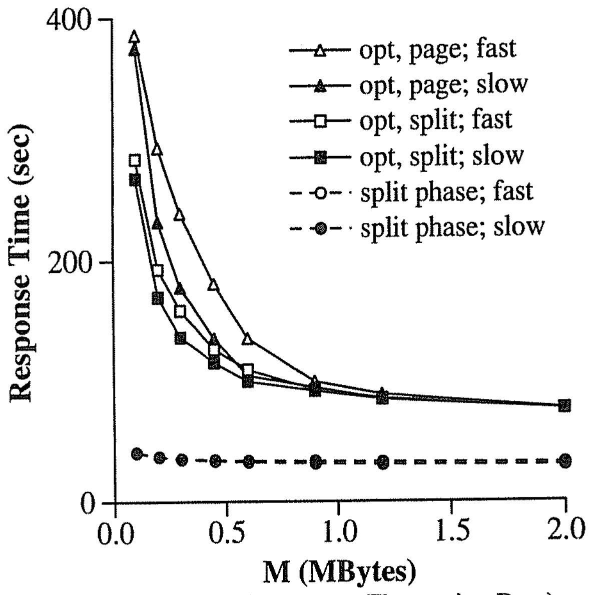

Figure 4.11: *repl*6 (Memory Fluctuation Rate)

To summarize, the results of this experiment lead us to conclude that, over a wide range of memory fluctuation rates, the algorithm *repl* 6,*opt*,*split* delivers the best overall performance among those that we considered. Dynamic splitting therefore appears to be a promising merge-phase adaptation strategy in practice.

4.4. Sort-Merge Joins

Sort-merge join is a join algorithm employed by many existing database systems. Although recent work has shown hash join to often be superior to sort-merge join in performing ad-hoc join operations [Brat84, DeWi84, Shap86], sort-merge join is still useful under certain conditions, e.g. when significant data skew is present, or when the results need to be presented in sorted order [Grae91]. Hence sort-merge join is likely to continue to be offered as one of the alternative join algorithms in future DBMSs. In this section, we address the issue of extending the techniques that we have explored earlier to handle sort-merge joins, thus making them memory-adaptive.

4.4.1. Memory-Adaptive Sort-Merge Joins

Like the external sort algorithm, a sort-merge join consists of a split phase and a merge phase. The split phase divides the two source relations into two separate sets of sorted runs. This is exactly as in the case of external sorts, except that now there is an additional relation to split. The in-memory sorting methods that we have examined, namely replacement selection, Quicksort and replacement selection with block writes, can thus be used here without any changes. In the merge phase, runs from both relations are merged concurrently, and sorted tuples from the two relations are joined directly as they are merged. In the event that the total number of runs from the two relations exceeds the available memory, the final merge step, i.e. the step that combines all the runs from both relations and produces the join results, has to be split. The preliminary step that is created as a result of this split will work on one of the relations, merging some of its existing runs into a longer sorted run. Since there are two relations, the preliminary step has a choice of which relation to merge. To minimize the cost of the preliminary step, the chosen relation is the one that will lead to a smaller total input size for the merge step. For example, if the preliminary step has to merge 15 runs, the number of pages in the smallest 15 runs from each individual relation is summed, and the relation with the smaller sum is selected. Any of the three merge-phase adaptation strategies, i.e. suspension, paging and dynamic splitting, can be used to adapt the sort-merge join to memory fluctuations during the merge phase. However, the naive and optimized merging strategies have to be modified slightly in order to comply with the requirement that each preliminary step merges only runs from the same relation.

During a split, the desired number of runs to be merged in the preliminary step is determined by either the naive or the optimized merging strategy. In some cases, one or both of the relations may not have that many runs. To illustrate this point, consider a situation where a sort-merge join has 11 buffers, and the two relations are split into 5 runs and 14 runs, respectively. Both naive merging and optimized merging will attempt to merge 10 runs in the preliminary step so that the remaining runs can be merged all at the same time. Unfortunately, the first relation has only 5 runs, so it cannot be chosen for the preliminary step. In such cases, we modify the naive and optimized merging strategies to select the relation that has more runs for the preliminary step, in order not to introduce more steps to the merge phase.

4.4.2. Experiment and Results

Since the same basic mechanisms work for both external sorts and sort-merge joins, we expect the relative performance trade-offs between the different in-memory sorting methods, merging strategies and merge-phase adaptation strategies to be the same in both cases. To confirm this, we present one set of experimental results. In this experiment, each sort-merge join involves two relations, **R** and **S**, of sizes $||R||$ and $||S||$, respectively. We let $||R||$ be 256 pages (2 MBytes) and $||S||$ be 2560 pages (20 MBytes) to simulate a primary key-foreign key join. Moreover, **M**, the total number of buffers, is varied while the other parameters are kept constant at their settings of the baseline experiment.

The join response times for this experiment are plotted in Figures 4.12 and 4.13. As expected, the performance trends for each algorithm, as well as the relative trade-offs between the different algorithms, are virtually identical to what we saw for external sorts in Figures 4.6 and 4.7: Dynamic splitting is clearly the merge-phase adaptation strategy of choice, while replacement selection with block writes is the winner among the in-memory sorting methods. Moreover, optimized merging outperforms naive merging.

In summary, the results of this experiment confirm that the merge-phase adaptation strategies, in-memory sorting methods and merging strategies that we have explored in the context of external sorts are

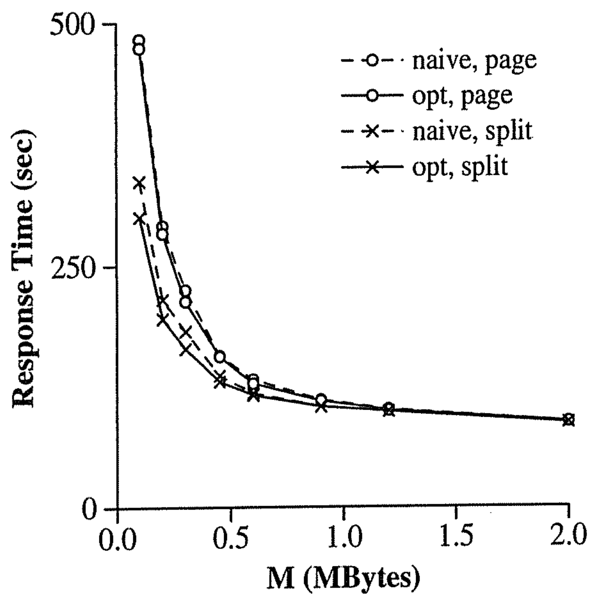

Figure 4.12: *repl*6 (Sort-Merge Joins)

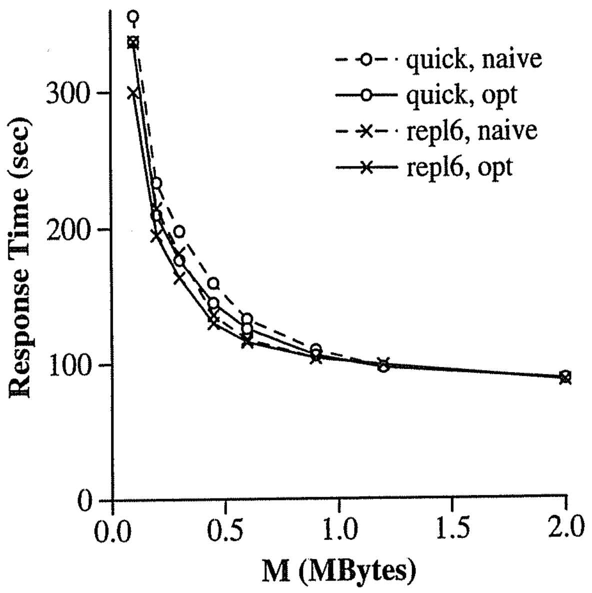

Figure 4.13: *split* (Sort-Merge Joins)

equally applicable to sort-merge joins. Therefore the combination of *repl*6,*opt*,*split* provides an effective means to do sort-merge joins in the face of fluctuations in memory availability.

4.5. Conclusion

In this chapter, we have addressed the problem of performing external sorts in situations where the amount of memory available to a query may be reduced or increased during its lifetime. Since external sorts typically require large numbers of buffers to execute efficiently, they are very susceptible to fluctuations in memory availability. Simple approaches that react to a reduction in an external sort's allocated memory by suspending the sort altogether, or by paging the buffers of the sort into and out of the remaining memory, may lead to under-utilization of system resources or thrashing. Furthermore, these approaches do not allow external sorts to make use of extra memory (beyond their initial memory allocation) that may become available during their lifetime. There is therefore a need, as we saw for hash joins in Chapter 3, for more sophisticated approaches that enable external sorts to adapt to memory fluctuations.

An external sort consists of two phases: the split phase fetches portions of the relation into memory, where they are sorted and then written out as sorted runs, and the merge phase combines the resulting runs into the sorted result. The merge phase consists of one or more merge steps, each of which combines a number of runs into a single, longer run. We studied Quicksort and replacement selection, two common in-memory sorting methods that are used for the split phase. In addition, we studied a variation of replacement selection that uses block writes to reduce disk seeks. All three in-memory sorting methods allow external sorts to respond to memory shortages by writing sorted tuples out to reduce their buffer usage; when memory increases, the newly allocated memory is used to fill more relation pages. In contrast to the in-memory sorting methods, the merge phase is not as easily adapted to memory fluctuations. We therefore examined hybrid approaches that allow external sorts to adapt to memory fluctuations only in the split phase, letting the database management system suspend the external sorts or page their buffers if memory shortages occur while they are in the merge phase. In addition, we proposed a merge-phase adaptation strategy, called *dynamic splitting*, that enables external sorts to better respond to memory shortages and to exploit excess memory in the merge phase by involving the sorts in adapting to memory fluctuations. This strategy splits an executing merge step into sub-steps that fit within the remaining memory when a shortage occurs, and it combines existing merge steps into larger steps (i.e. steps that merge more runs at once) to take advantage of excess buffers when they become available.

To understand how effective the different in-memory sorting methods and merge-phase adaptation strategies are in dealing with memory fluctuations, we undertook a series of experiments using the detailed simulation model described in Chapter 2. A series of experiments revealed that, when the available memory is small relative to the relation to be sorted, the merge-phase adaptation strategy is the dominant performance factor. Among the merge-phase adaptation strategies, dynamic splitting outperforms paging; the smaller the size of memory is relative to the relation, the more significant the performance difference between the two strategies becomes. The third merge-phase adaptation strategy, suspension, consistently yields unsatisfactory response times. Thus dynamic splitting appears to be an attractive strategy for sorting large relations. Our results also showed that replacement selection with block writes is the preferred in-memory sorting method since it consistently produced response times that were at least as fast as Quicksort. Overall, our results indicate that the combination of dynamic splitting and replacement selection with block writes enables external sorts to deal effectively with memory fluctuations.

Like external sorts, sort-merge joins are vulnerable to memory fluctuations due to their large memory requirements. The sort-merge join algorithm is also made up of a split phase and a merge phase. The split phase divides each of the two operand relations into sets of sorted runs. In the merge phase, runs from both relations are combined concurrently, and the sorted tuples from the two relations are joined directly. If there are too many runs to be merged at the same time, preliminary steps are created to merged some of the existing runs from one (or both) relation(s) into longer sorted runs. The same techniques that we examined in the context of external sorts can be applied to sort-merge joins in order to make them memory-adaptive. Moreover, the same relative performance trade-offs apply to both external sorts and sort-merge joins. We will therefore adopt the combination of dynamic splitting and replacement selection with block writes to process both external sorts and sort-merge joins in the second half of this thesis, which studies higher-level query scheduling issues.

CHAPTER 5

MANAGING MEMORY FOR REAL-TIME QUERIES

In Chapters 3 and 4, we were concerned with developing techniques to allow queries to execute efficiently in the face of memory fluctuations. In particular, we identified Partially Preemptible Hash Join (PPHJ) with late contraction, expansion, and priority spooling as the memory-adaptive technique of choice for hash joins. For external sorting, we identified the combination of dynamic splitting and replacement selection with block writes as the technique of choice. With these low-level query primitives in place, we are now ready to move on to higher-level query scheduling issues.

As discussed in Chapter 1, real-time database systems (RTDBS) need to employ multiprogramming so that all of their resources can be utilized productively to service incoming queries. However, admitting too many queries at the same time can lead to thrashing, making high concurrency harmful instead of helpful. Multiprogramming is therefore a two-edged sword, and RTDBSs require an admission control mechanism to protect them against thrashing. Once the degree of multiprogramming has been determined, another important issue must be addressed, i.e., how much memory to give each admitted query. These are the concerns that we now turn our attention to.

In this chapter, we introduce a *Priority Memory Management* (PMM) algorithm that is designed to schedule queries in firm RTDBSs. PMM does not assume any advance knowledge of workload characteristics or query execution times, as such knowledge is usually not available in a database system. Instead, the PMM algorithm controls the number of queries that may gain admission at any given time by dynamically choosing a target multiprogramming level (MPL) to balance the demands on the system's memory, CPU, and disks. Moreover, PMM can either insist that queries be admitted only with their maximum memory allocations, or it can give higher-priority queries their maximum required memory while allowing lower-priority queries to run with their minimum requirements. Both the target MPL and the memory allocation policy are chosen based on past system behavior. The Earliest Deadline policy [Liu73], which gives higher priority to queries whose deadlines are more imminent, is used to guide the admission and memory allocation decisions of PMM.

5.1. Related Scheduling Work

While a number of studies have addressed real-time transaction scheduling [eg., Abbo88b, Hari90a, Huan89] and disk scheduling [Abbo89, Abbo90, Care89, Chen91, Kim91], to the best of our knowledge no work has dealt with query scheduling issues in RTDBSs. The work that is most relevant to our work here is reported in [Corn89, Yu93]. In that work, the authors examined the effect of memory allocations on query response times in traditional (non-real-time) database systems, and they concluded 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 incorporated in the memory allocation strategies of PMM.

5.2. Priority Memory Management

In firm real-time database systems [Hari90a], queries become worthless if they fail to complete by their deadlines. Consequently, the primary performance objective of an RTDBS is to minimize the number of missed deadlines without intentionally discriminating against any particular type of queries. In order to achieve this objective, resource scheduling decisions in these systems have to be priority-driven. The Priority Memory Management (PMM) algorithm is a priority-cognizant algorithm designed to regulate memory usage for firm real-time query workloads.

The PMM algorithm consists of an admission control component and a memory allocation component. Both components employ the Earliest Deadline (ED) scheduling policy [Liu73], so queries that are more urgent are given higher priority in admission and memory allocation decisions than queries whose deadlines are further away. The ED policy is adopted here, instead of policies that take into account query execution times, because (accurate) execution time information is usually not available a priori in a database system. The admission control component sets the target multiprogramming level (MPL) by statistical projection from past miss ratios and their associated MPL values. In cases where the statistical projection 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 5.1.

5.2.1. 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.

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

Table 5.1: PMM Algorithm Parameters

5.2.1.1. 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 [Drap81]. 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, $\Sigma\, mpl_i$, $\Sigma\, mpl_i^2$, $\Sigma\, mpl_i^3$, 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. 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 monotonic 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 sticks to the MPL that the miss ratio projection method picked.

Type 3: *The curve is monotonic 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 5.2.3).

5.2.1.2. 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 under-utilized 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, one of the values 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 $<util_i, mpl_i>$ of observed utilization values and their associated MPLs by using the least squares method [Drap81], 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, $\Sigma\ mpl_i$, $\Sigma\ mpl_i^2$, $\Sigma\ util_i$, and $\Sigma\ mpl_i \times util_i$, where k denotes the number of times PMM is invoked.

An alternative to using the formula given above to determine MPL_{New} would have been to simply choose the MPL value on the fitted line that corresponds to the desired utilization level. The drawback of this alternative is that, due to workload fluctuations, the fitted line may not reflect the true relationship between MPL and utilization very well. This is especially a problem at the start, where few statistics are available, and where, unfortunately, PMM has to rely on the RU heuristic because it does not yet have sufficient statistical data to apply the miss ratio projection method. We therefore ruled out this alternative from further consideration.

5.2.2. 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 [Shap86], 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 [Shap86].

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 of PMM 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 then proportional allocation (as was shown in [Corn89, Yu93]); this will be verified quantitatively in Section 5.3.1.

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. Consequently, at the end of this memory allocation process, 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. The initial variations are the reason why we require the dynamic query processing techniques described in the preceding two chapters.

The Max strategy, by insisting on the maximum memory allocation, eliminates the 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 sets the target MPL in this mode to ∞, admitting as many 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 non-zero 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 [Devo91] 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.

5.2.3. Dealing with Workload Changes

PMM attempts to minimize query miss ratios by tailoring its MPL setting and memory allocation strategy 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 $Change_{ConfLevel}$ [Devo91] 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 5.1) to reduce the chances of PMM wrongly reacting to inherent workload fluctuations.

5.2.4. An Example

Having presented the PMM algorithm in detail, we now finish by illustrating it with a simple example. Suppose that the first batch of *SampleSize* queries produces point *a* in Figure 5.1(a) 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 the 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 5.2.1.1) in Figure 5.1(a). This curve causes PMM to experiment with an even higher MPL, the consequence of which is indicated by point *d* in Figure 5.1(b). Applying the projection method again, PMM now

[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.

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.

5.3. Experiments and Results

In this section, our database system simulator will be used to evaluate the performance of the Priority Memory Management (PMM) algorithm. For comparison purposes, we shall also examine three static memory allocation algorithms: Max, MinMax-N, and Proportional-N. The Max algorithm always employs the Max strategy in its memory allocation decisions. MinMax-N admits the N highest-priority queries, dividing the available memory among these N queries according to the MinMax policy. A special case of MinMax-N is MinMax-∞, which admits as many queries as the available memory allows by not explicitly limiting the MPL. In this section, MinMax-∞ will be frequently used to compare against PMM, so we shall refer to MinMax-∞ simply as MinMax. Note that PMM is an adaptive algorithm that dynamically chooses between the Max algorithm and the MinMax-N algorithm, where N is the target MPL setting. The final algorithm to which PMM will be compared, Proportional-N, behaves like MinMax-N, except that Proportional-N gives the N admitted queries the same percentage of their maximum buffer requirements subject to the

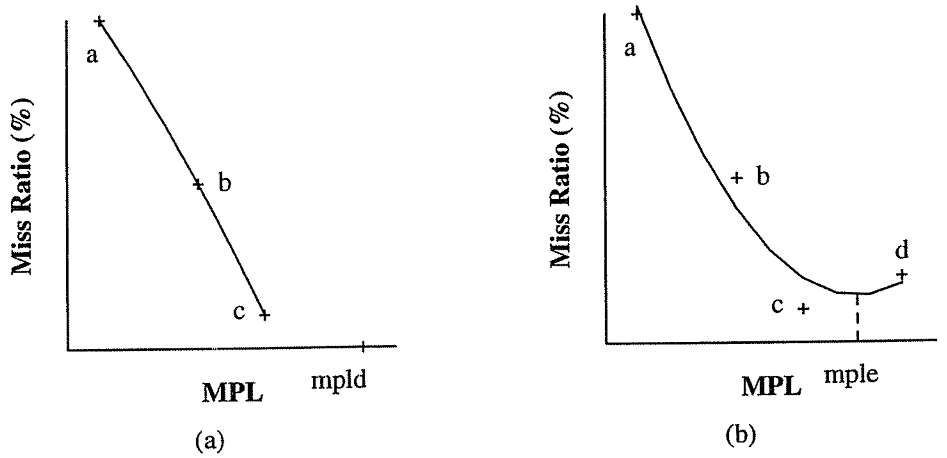

Figure 5.1: Admission Control Decision Making

condition that the memory allocation of an admitted query must at least equal, if not exceed, its minimum requirement. As in the case of MinMax, we shall simply refer to Proportional-∞ as Proportional. For ease of reference, the various algorithms are listed in Table 5.2.

We will begin our evaluation of PMM with a baseline experiment, with further experiments being carried out by varying a few parameters each time. The performance metric of interest here is the average query miss ratio, which is the percentage of queries that the system fails to complete by their deadlines. Unless stated otherwise, each experiment was run for 10 hours of simulated time, allowing a minimum of 2000 query completions. We also verified that the size of the 90% confidence intervals for miss ratios (computed using the batch means approach [Sarg76]) was within a few percent of the mean in almost all cases, thus ensuring that our results are statistically valid.

5.3.1. Baseline Experiment

In the first experiment, 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 2560 pages (20 MBytes). The rest of the resource parameters are kept at their settings of Table 3.3. 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 1800 pages and $||S||$ is selected from the range [3000, 9000] pages. Moreover, the slack ratio interval is set to [2.5, 7.5]. The database and workload parameters are summarized in Table 5.3.

Figure 5.2 plots the miss ratios for Max, MinMax, Proportional, and PMM as a function of the arrival rate. The figure shows that MinMax consistently delivers the lowest miss ratio for this experiment, followed

Indicator	**Algorithm**
Max	Max algorithm
MinMax-N	MinMax algorithm with an MPL limit of N
MinMax	MinMax algorithm with no MPL limit
Proportional-N	Proportional algorithm with an MPL limit of N
Proportional	Proportional algorithm with no MPL limit

Table 5.2: Algorithms for Comparison with PMM

Database	Meaning	Setting
NumGroups	Number of relation groups in the database	2
$RelPerDisk_1$	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
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

Table 5.3: Database and Workload Parameter Settings for Baseline Experiment

very closely by PMM. Proportional performs satisfactorily initially, achieving a near 0% miss ratio at λ = 0.04 queries/second. As the arrival rate increases, however, the performance of Proportional deteriorates rapidly until, at λ = 0.08 queries/second, Proportional produces a hefty 25% miss ratio, which is almost double that of MinMax and PMM. The worst algorithm is Max, which manages to match the performance of Proportional only under lighter load conditions. As the workload mounts, Max degenerates even faster than Proportional, missing four times as many deadlines as MinMax and PMM. 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 four algorithms, we shall analyze each algorithm in turn with the aid of Figures 5.3 and 5.4, which give the disk utilizations and average observed MPLs (as opposed to the target MPL set by PMM, which serves to limit the maximum MPL in the system) respectively, and Table 5.4, which lists the admission waiting time, execution time and total response time for the various algorithms.

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 2 queries to be admitted at the same time (see Figure 5.4) since each query requires an average of 1321 buffers ($F\times$1200 pages for **R** plus one I/O buffer). This makes memory the bottleneck for Max, as evidenced by the high admission waiting times recorded in Table 5.4. 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

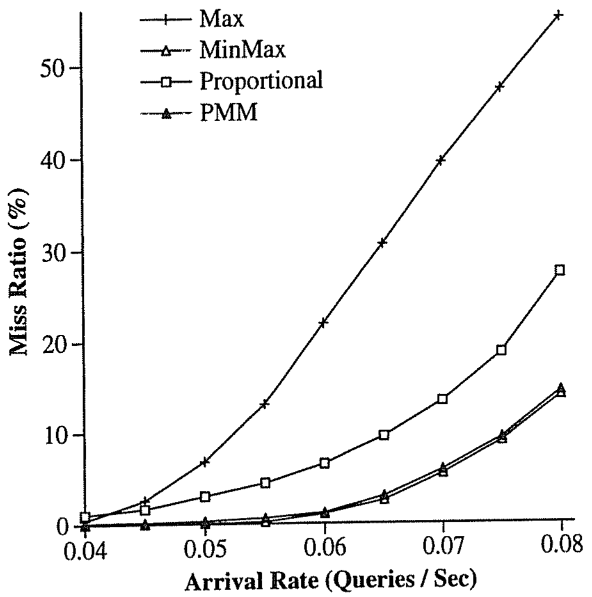

Figure 5.2: Miss Ratio (Baseline)

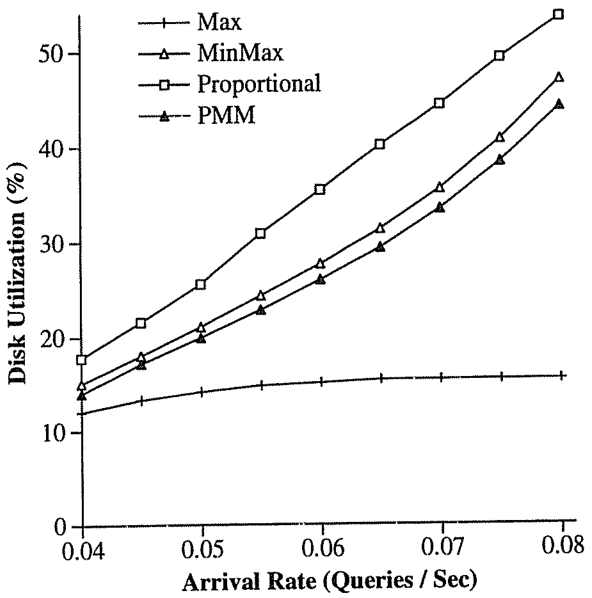

Figure 5.3: Disk Utilization (Baseline)

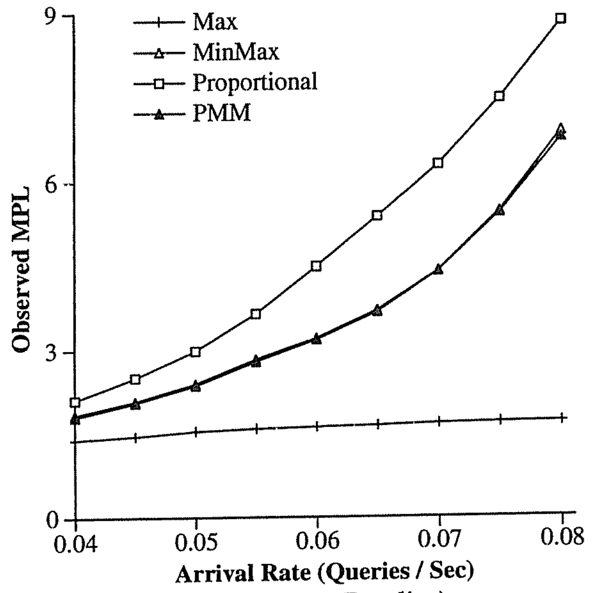

Figure 5.4: MPL (Baseline)

Arrival Rate	0.040	0.045	0.050	0.055	0.060	0.065	0.070	0.075	0.080
Max									
Waiting	12.4	22.4	36.4	57.2	81.4	97.6	107.3	113.5	117.3
Execution	39.5	37.9	35.4	34.5	32.9	28.1	25.9	23.9	22.4
Total	51.9	60.3	71.8	91.7	114.3	125.7	133.2	137.4	139.7
MinMax									
Waiting	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Execution	40.9	43.2	45.5	49.2	53.1	59.6	68.3	78.8	92.1
Total	40.9	43.2	45.5	49.2	53.1	59.6	68.3	78.8	92.1
Proportional									
Waiting	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Execution	52.9	56.0	61.2	67.3	75.8	84.0	92.4	99.7	110.8
Total	52.9	56.0	61.2	67.3	75.8	84.0	92.4	99.7	110.8
PMM									
Waiting	3.0	3.2	3.3	3.5	3.7	3.8	3.9	4.0	4.0
Execution	40.2	42.3	45.1	48.2	52.5	58.5	66.3	76.4	89.4
Total	43.2	45.5	48.4	51.7	56.2	62.3	70.2	80.4	93.4

Table 5.4: Average Timings (seconds) for Baseline Experiment

rate increases from 0.04 to 0.08 queries/second, which explains why, unlike the other three 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 Figure 5.4 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/second, 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. As shown in Table 5.4, 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 Figure 5.2.

Like MinMax, Proportional attempts to reduce query response times by not insisting on maximum memory allocation as an admission criterion. This is why Proportional also produces higher MPLs than Max.

The difference between Proportional and MinMax is that Proportional divides up the available memory among the admitted queries in proportion to their demands, rather than running low-priority queries with minimum allocations while giving high-priority queries their maximum required memory (as in MinMax). Unfortunately, the faster execution times that the low-priority queries enjoy from receiving more than their minimum required memory are overwhelmed by the execution time penalty that the high-priority queries pay as a result of being forced to run with less-than-maximum memory allocations. The average execution time that Proportional produces is therefore higher than that of MinMax. The longer query execution times also cause an increase in the number of queries that are running concurrently, as shown in Figure 5.4, which in turn reduces the amount of memory that each query receives. This increases the queries' reliance on the CPU and disks, resulting in further increases in the queries' execution times. Consequently, Proportional utilizes memory much less effectively than MinMax. As mentioned earlier, similar observations about the inferiority of Proportional-style policies were made in [Corn89, Yu93] in a non-real-time context.

We now turn our attention to the PMM algorithm. In order to understand how PMM adapts itself to the workload, we examine Figure 5.5, which traces the target MPL settings of PMM over the initial 10 hours of operation at an arrival rate of 0.075 queries/second. PMM 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 PMM 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 PMM 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, as the low 4-second admission waiting time in Table 5.4 suggests. Indeed, Figure 5.4 shows that PMM consistently achieves high MPL settings, thus enabling it to behave like the Min-Max algorithm. This is why PMM manages to closely match the performance of MinMax, which offers the best miss ratios for this experiment.

Having studied the relative performance trade-offs of the memory allocation algorithms, we now briefly examine the demand that these algorithms place on the system's underlying memory-adaptive query processing primitives. Figure 5.6 shows, as a function of the arrival rate, the average number of times that a query's memory allocation changes under each of the memory allocation algorithms. The Max algorithm does not require queries to adapt to memory fluctuations, as it only executes queries with their maximum required memory (suspending them otherwise). In contrast, the other three algorithms do expose executing queries to changes in their memory allocations. Under MinMax (and hence PMM, since it mimics the MinMax algorithm in this experiment), the allocation of a query may vary anywhere from its minimum to its maximum memory requirements initially, gradually stabilizing at the maximum only as its deadline draws near. The number of memory fluctuations that a query experiences during this initial period is determined by the frequency at which higher-priority queries enter and leave the system, which explains why the number of memory fluctuations increases with the arrival rate (growing from a mere 5 changes per query at an arrival rate of 0.04 queries/second to an average of 10 fluctuations per query at 0.08 queries/second). The algorithm that generates the most memory fluctuations is Proportional, which produces up to 2.5 times as many memory fluctuations per query as MinMax. This occurs because Proportional always distributes memory proportionally among all admitted queries, therefore subjecting queries to memory changes throughout their entire lifetimes.

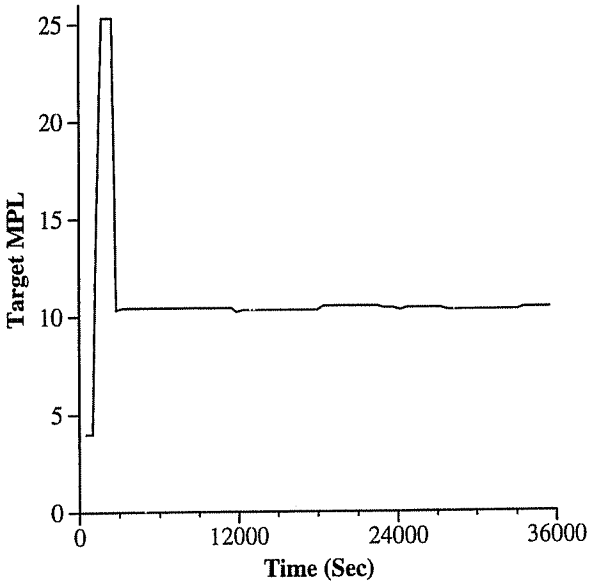

Figure 5.5: PMM MPL, λ = 0.075 (Baseline)

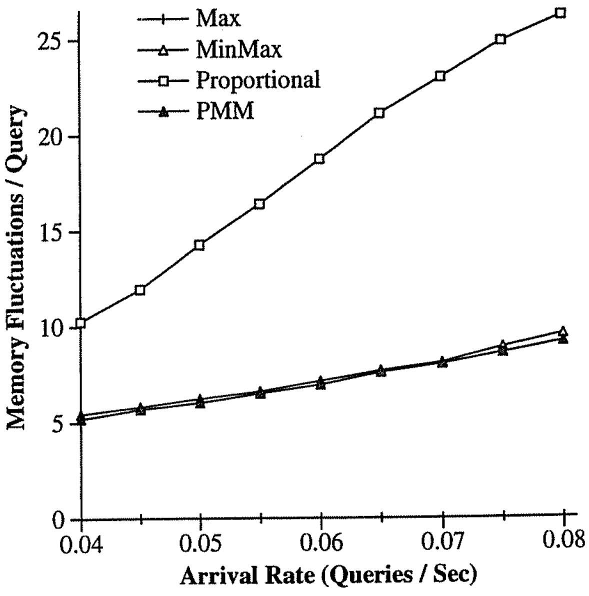

Figure 5.6: Memory Fluctuations (Baseline)

To summarize the results of this experiment, we can derive the following conclusions about situations where memory is the bottleneck resource of an RTDBS: First, insisting on maximum memory allocation as an admission criterion is undesirable. Instead, an RTDBS needs to be willing to run queries at memory allocations that are below their maximum requirements so that enough queries can be admitted to take advantage of the RTDBS's disk and CPU resources. This is facilitated by memory-adaptive query processing techniques (such as those of [Pang93a, Pang93b]) that permit queries to execute efficiently in the face of memory fluctuations. Among the algorithms that do not insist on maximum memory allocations, Proportional allocation leads to very large miss ratios and should be avoided. This is why PMM employs MinMax, instead of Proportional, allocation, when it detects that running queries with sub-maximal memory allocations is beneficial. Finally, PMM seems to be capable of finding the right MPL setting and memory allocation strategy within a few iterations, achieving low query miss ratios by balancing the load on the system's various resources.

5.3.2. Moderate Disk Contention

In the next experiment, we investigate how PMM performs when disk contention becomes more of a consideration in memory allocation decisions, though memory is still the bottleneck resource. The number of disks is reduced here to 6, while the rest of the parameters remain at their settings from the baseline experiment. We will exclude the Proportional algorithm since it has already been demonstrated to be inferior to MinMax. The performance statistics for the remaining three algorithms, Max, MinMax-N and PMM, are given in Figures 5.7, 5.8 and 5.9, which plot as a function of the arrival rate their miss ratios, disk utilizations, and observed MPLs, respectively. These figures show that the behavior of Max is essentially the same as in the baseline experiment. We shall therefore not discuss Max here, instead focusing on MinMax and PMM, both of whose behaviors differ significantly from those observed previously.

We first analyze the performance of the MinMax algorithm. Figure 5.7 shows that MinMax no longer provides the best performance. In fact, MinMax now misses many more deadlines than PMM when the system is heavily loaded. The performance deterioration of MinMax here is due to its unrestrained admission policy. In this experiment, where disk contention is not negligible, the system does not always have enough disk capacity for all of the queries that MinMax admits. This is evidenced by the higher average disk utiliza-

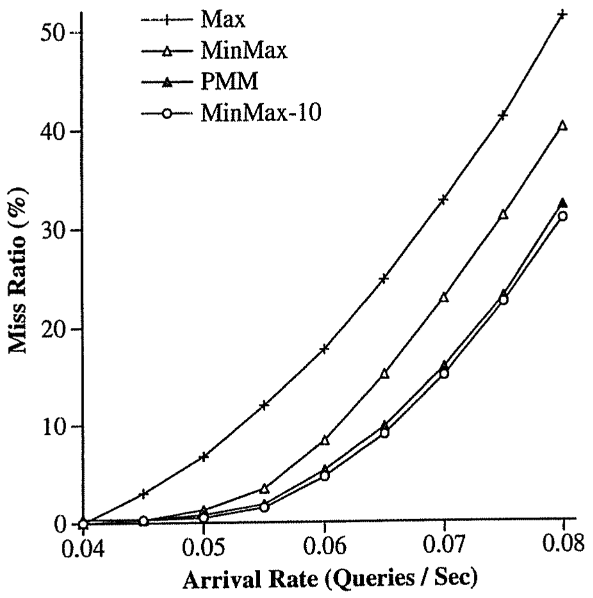

Figure 5.7: Miss Ratio (Disk Contention)

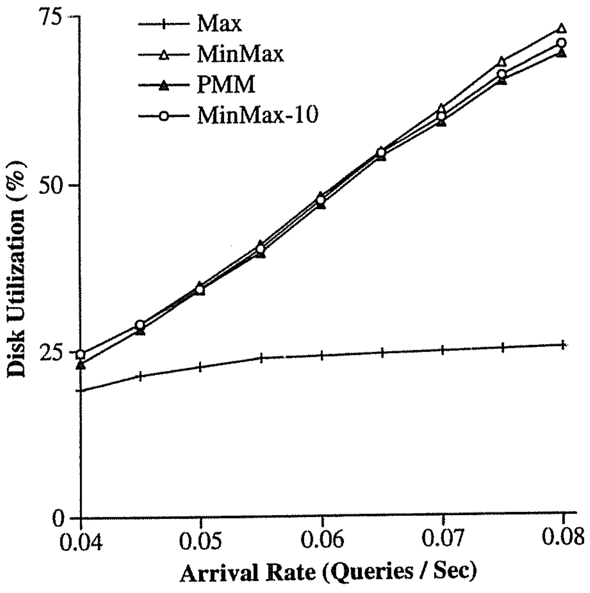

Figure 5.8: Disk Utilization (Disk Contention)

tions in Figure 5.8, which exceed 70% under heavy load conditions. As a result, some of the low-priority queries remain essentially inactive even after being allotted memory because they do not get the opportunity to access the disks under the priority disk scheduling policy. This unproductive use of memory unnecessarily forces higher-priority queries to run below their maximum memory allocations and increases their dependence on the CPU and disks, resulting in the observed rise in MinMax's miss ratios.

Since MinMax's unsatisfactory performance in this experiment stems from its unrestrained admission policy, we must examine other MinMax-N variants in order to explain PMM's performance. Figure 5.10 plots the miss ratios produced by MinMax-N as a function of N for an arrival rate of 0.07 queries/second. The MinMax-N variants that are included in this figure cover the entire spectrum of trade-offs. At one end of the spectrum, the MinMax-N algorithms with low N values are similar to Max, as every admitted query is able to run with maximum memory allocation due to the low MPL settings. At the other end of the spectrum is MinMax-20, which essentially performs like MinMax (not shown)[2]. Figure 5.10 shows that the best

[2] Theoretically, MinMax allows up to an average MPL of 69 for this workload. In practice, the chances of having more than 20 queries in the system at the same time here is so rare that, for all practical purposes, MinMax-20 is the same as MinMax.

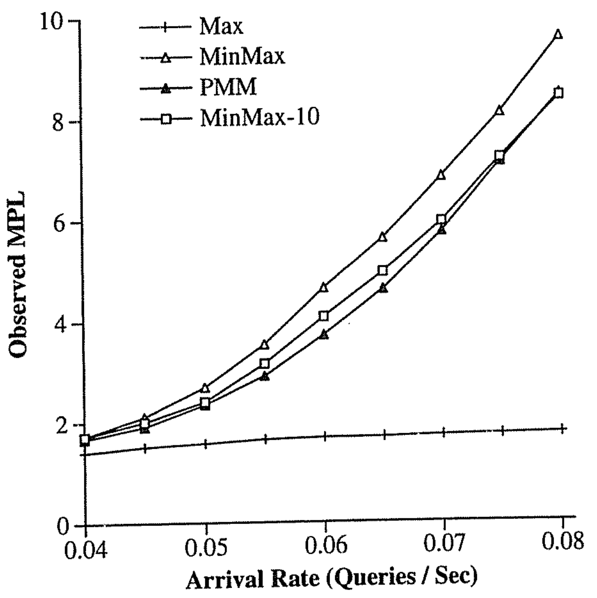

Figure 5.9: MPL (Disk Contention)

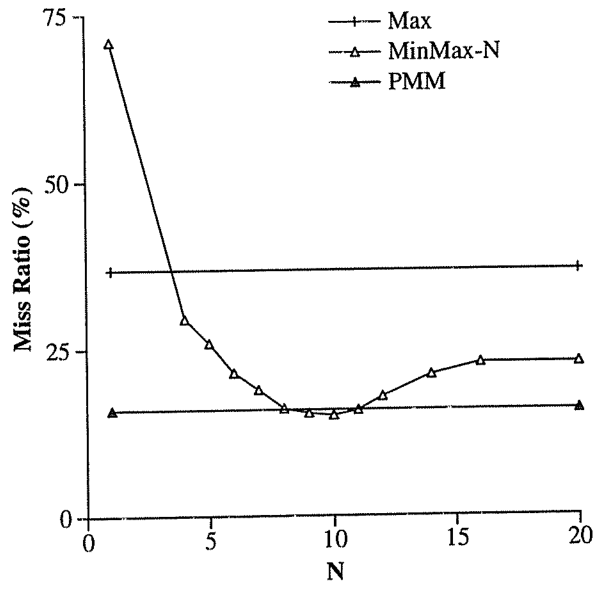

Figure 5.10: MinMax-N, $\lambda = 0.07$ (Disk Contention)

performance for this workload is achieved by MinMax-10, which utilizes the CPU and disks much more effectively than Max by admitting more queries into the system — but not so many queries that thrashing occurs, as is the problem with the unrestrained approach of MinMax. We also conducted a series of experiments like Figure 5.10 at other arrival rates, and the results of those experiments unanimously confirmed that MinMax-10 indeed delivers the best performance for the present workload.

Having identified MinMax-10 as the best MinMax-N algorithm for this experiment, we now proceed to evaluate PMM against MinMax-10. The curves in Figure 5.9 show that the observed average MPLs for PMM remain consistently close to those of MinMax-10. This indicates that the MPL search mechanism of PMM succeeds at bringing its MPL setting to the proximity of the best MPL value, which explains why PMM outperforms both Max and MinMax. In fact, Figure 5.7 shows that PMM manages to meet almost as many deadlines as MinMax-10 over the entire range of arrival rates that we investigate, delivering miss ratios that are worse than those of MinMax-10 by at most 2%.

The results of this experiment show that, while Max leads to under-utilization of the CPU and disks in memory-constrained situations, MinMax can produce thrashing when disk contention is not negligible. Therefore, some trade-off between Max and MinMax has to be reached, i.e., a MinMax-N algorithm is needed. Since the best MinMax-N algorithm depends on the system configuration and workload characteristics, which are usually not known in advance, the right MinMax-N algorithm to employ has to be dynamically selected. PMM demonstrated its ability here to quickly find the appropriate MinMax-N algorithm by steering itself to the best MPL setting.

5.3.3. Workload Changes

The first two experiments lead us to the conclusion that PMM performs well for relatively stable real-time workloads. The objective of this experiment is to find out how quickly PMM can adapt to workload changes. This is achieved by subjecting the various memory allocation algorithms to a workload that alternates between two classes of hash joins, Small and Medium, every 2 to 5 simulated hours. For the Small class, $||R||$ ranges between 50 and 150 pages, while $||S||$ ranges from 250 to 750 pages. The characteristics of the Medium class are the same as those of the baseline workload. These two classes pose different demands on the system's resources. On one hand, it takes an average of only 111 memory pages to satisfy the maximum demand of each hash join from the Small class. Thus the disks, rather than the memory, are the bottleneck for the Small class, and the Max algorithm is therefore appropriate for this class. On the other hand, the system is memory-constrained with the Medium class, making a MinMax-N algorithm more desirable, as we saw previously. In order to highlight the performance trade-offs between the various algorithms, the arrival rates of the two classes are chosen so that the RTDBS is forced to operate under relatively heavy load conditions. The database and workload parameters are listed in Table 5.5. For this experiment, the number of disks is again set to 6, with the rest of the resource parameters set as in the baseline experiment.

Figures 5.11, 5.12, and 5.13 display the miss ratios of the three algorithms as a function of time, while Figure 5.14 traces the observed MPL under PMM. Figures 5.11 to 5.13 also give the average miss ratio over each interval along the top of each figure. Comparing the two static algorithms, we notice that MinMax's unrestrained admission policy again causes it to perform poorly: Whereas Max produces average miss ratios

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
$RelPerDisk_3$	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	0.07
$SRInterval_1$	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	2.8
$SRInterval_2$	Range of slack ratios for class *2* queries	[2.5, 7.5]
F	Fudge factor for hash joins	1.1

Table 5.5: Database and Workload Parameter Settings (Workload Changes)

of 16% and 33% for the Small and Medium classes[3], respectively, MinMax produces average miss ratios of 37% and 23% for the two classes. In contrast to MinMax, PMM is able to capitalize on the system's disk and CPU resources without suffering from thrashing. By dynamically selecting its MPL setting and memory allocation strategy based on the workload characteristics, PMM outperforms both Max and MinMax for the Medium class, missing only 15% of its queries on the average. Moreover, PMM successfully detects workload changes, switching back to Max mode for the Small class, so its average miss ratio for Small queries is just as low as that of the Max algorithm. Similar experiments under lighter loads revealed essentially the same trade-offs between the three algorithms; while the magnitudes of the differences were smaller there, the relative performance of the algorithms was the same as that seen here. We therefore conclude that PMM not only performs well under stable workloads, but is also capable of adapting to workload changes.

[3] The average miss ratio of the Medium class is derived by averaging the miss ratios over the three time intervals where the workload is made up of Medium queries.

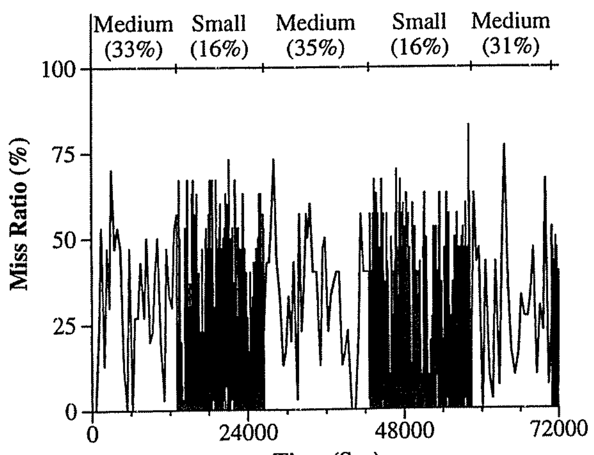

Figure 5.11: Max Miss Ratio (Workload Changes)

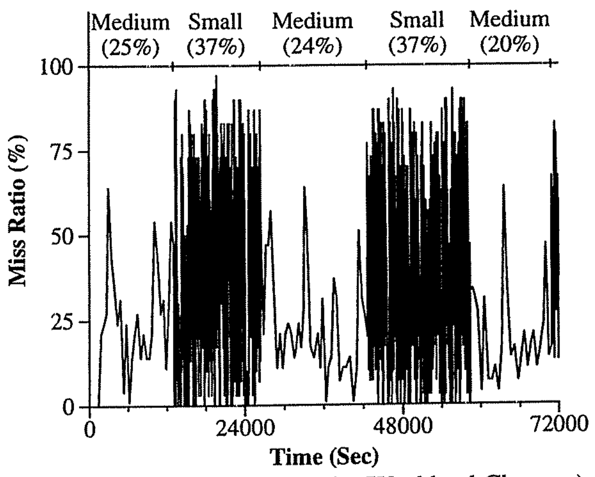

Figure 5.12: MinMax Miss Ratio (Workload Changes)

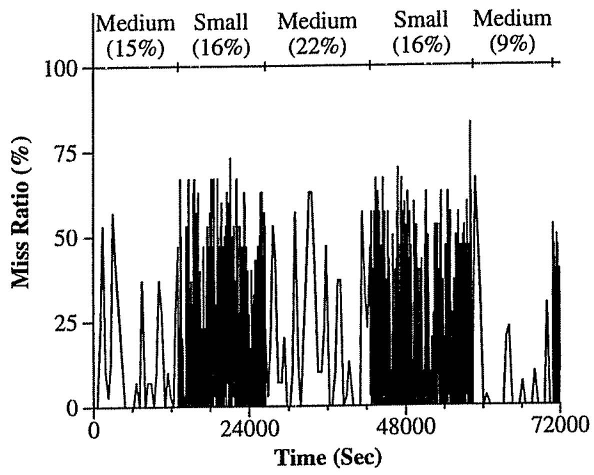

Figure 5.13: PMM Miss Ratio (Workload Changes)

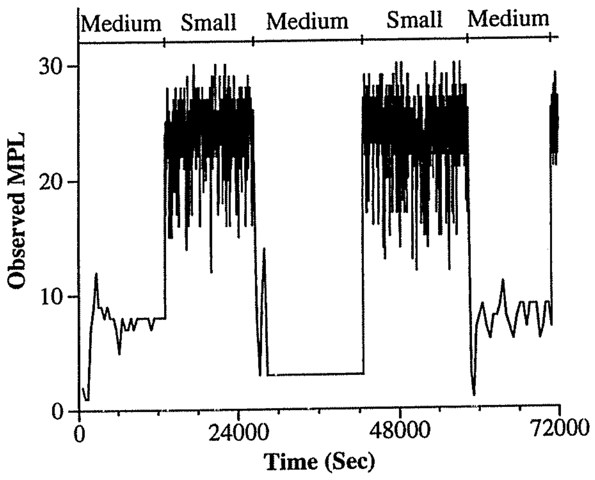

Figure 5.14: PMM MPL (Workload Changes)

5.3.4. Desirable Resource Utilization Levels

One of the input parameters of the PMM algorithm is the range of desirable resource utilization levels, $[Util_{Low}, Util_{High}]$. Until PMM has gathered sufficient statistics to estimate the MPL that leads to the lowest miss ratio, this input parameter is used to guide the MPL setting: If the observed resource utilization is below $Util_{Low}$, PMM increases its MPL, while an observed utilization in excess of $Util_{High}$ prompts PMM to lower its MPL setting. Up to this point, all of our experiments have used the range [0.70, 0.85] for this parameter. The choice of 0.85 for $Util_{High}$ is reasonable because, with resources being more than 85% utilized, the system most probably does not have enough capacity to service all of the admitted queries, so thrashing is likely to occur. The appropriate setting for $Util_{Low}$ is not as obvious, however. This experiment is designed to test the sensitivity of PMM to the $Util_{Low}$ setting. To do so, we vary $Util_{Low}$ from 0.50 to 0.80. The rest of the parameters are set as in the baseline experiment.

Figure 5.15 plots the miss ratios for PMM as a function of the arrival rate for different $Util_{Low}$ settings. This figure clearly shows that PMM delivers approximately the same performance over a wide range of $Util_{Low}$ values. This is not surprising, as PMM relies on the desirable resource utilization levels to set its MPL only during the initial period after startup. Since the precise value of $Util_{Low}$ does not matter, the default setting of 0.70 suffices.

5.3.5. Other Query Types

While we have demonstrated the capability of PMM for handling workloads that consist of hash joins, the PMM algorithm is designed to be a general memory management algorithm for RTDBSs; it is not limited to handling only hash joins. To verify that PMM is capable of handling other types of queries, we repeat the baseline experiment using external sorts. Each query in this new workload sorts a single relation **R**, where $||R||$ ranges from 600 to 1800 pages. All of the other workload and resource parameters (except arrival rates)

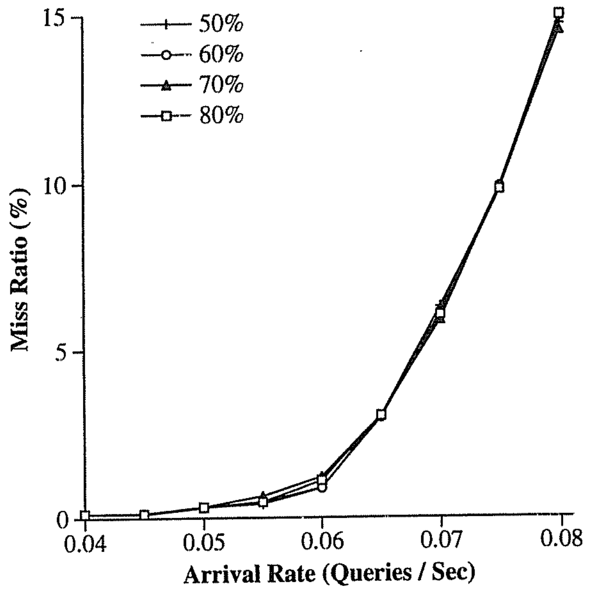

Figure 5.15: PMM Miss Ratio ($Util_{Low}$)

remain as they were in the baseline experiment. Here we include the Proportional algorithm once again for completeness of our evaluation.

The miss ratios of Max, MinMax, Proportional, and PMM for this workload are shown in Figure 5.16. Comparing this figure with Figure 5.2, we notice that Max performs much worse here than MinMax, Proportional and PMM. This is because the load that they place on the disks and CPU is lighter here, while the memory demands of the queries are about the same as before; on the average, each external sort only has to read in a 1200-page relation, whereas the average hash join in the baseline experiment had to deal with a 1200-page inner relation plus a 6000-page outer relation. Consequently, memory is a much more critical resource here, thus resulting in a situation that is even more favorable to the liberal admission policies employed by the other algorithms. Again, we see that PMM is able to select the appropriate MPL setting and allocation strategy, achieving miss ratios that are just about as low as those obtained by MinMax.

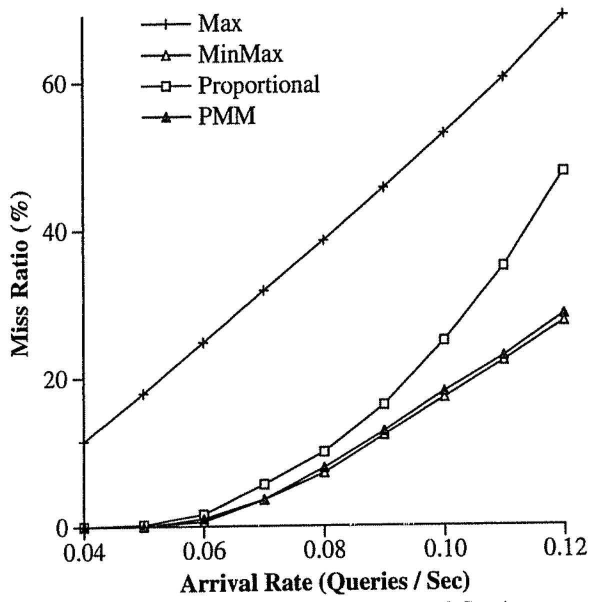

Figure 5.16: Miss Ratio (External Sort)

5.3.6. Multiclass Workload

Our last experiment is designed to study how the PMM algorithm performs when presented with a multiclass workload. We again simulate a workload that consists of two classes of hash joins, Small and Medium. The characteristics of the two classes are as listed in Table 5.5. However, instead of alternating between the two classes as in the "Workload Changes" experiment, here we activate both classes *together*. We fix the arrival rate of the Medium class at 0.065 queries/second and vary the arrival rate of the Small class. With the exception of the number of disks, which is raised to 12 to accommodate the heavier load here, the resource parameters remain as in the baseline experiment.

Figure 5.17 shows the overall *system* miss ratios produced by the Max, MinMax, and PMM algorithms. Interestingly, here the system miss ratio curve of PMM resembles that of MinMax initially, but gradually switches to follow that of Max as λ_{Small} increases. This behavior arises because PMM chooses its MPL and memory allocation strategy according to the average characteristics of the workload, which naturally affords the class that has a higher arrival rate a greater influence on its choices. Consequently, PMM adopts the MinMax strategy, which is more suitable for Medium queries, only when λ_{Small} is low. As λ_{Small} rises, PMM allows the increasing influence of Small queries to sway it to Max mode. While operating in this mode is very effective in minimizing the system miss ratio, as Figure 5.17 shows, it severely limits the MPL of the Medium class and causes a disproportionally large number of Medium queries to miss their deadlines. This bias is clearly evident in Figure 5.18, which plots the miss ratios of the individual classes. Such biased behavior may not be acceptable for certain applications. In the next chapter, we will examine the possibility of augmenting PMM with a mechanism to allow an RTDBS system administrator to specify the desired relative *class* miss ratios to support applications that require "fairer" real-time query services.

5.3.7. 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? In particular, would PMM still be able to choose appropriate MPL settings and memory allocation strategies quickly? To explore these

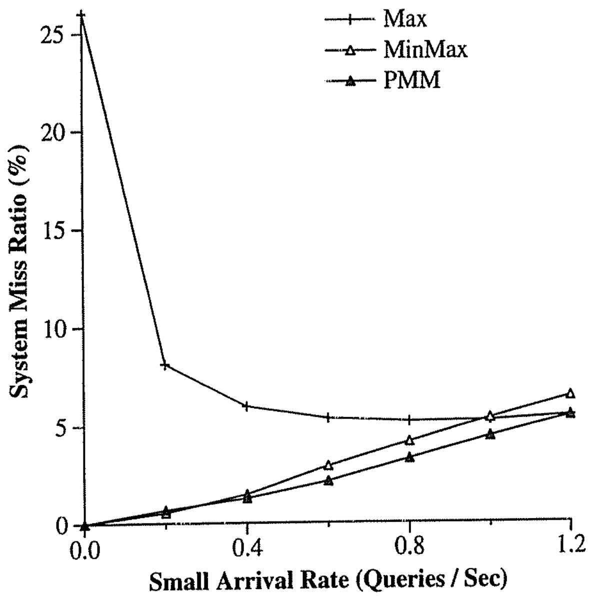

Figure 5.17: System Miss Ratio (Multiclass)

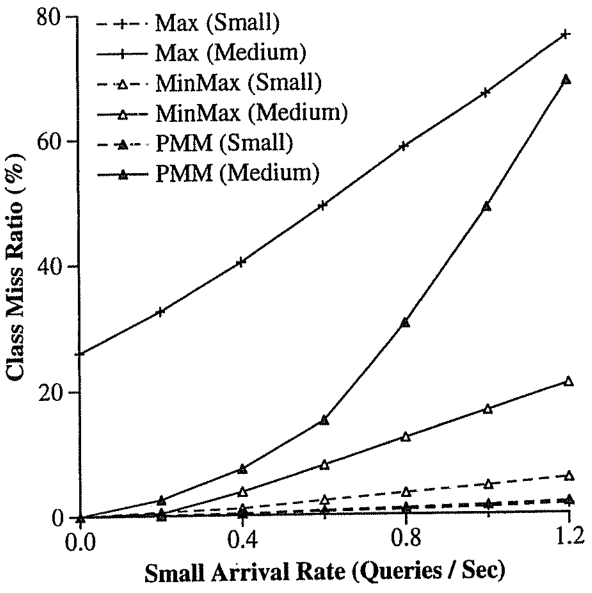

Figure 5.18: Class Miss Ratio (Multiclass)

issues, let us consider a scenario with the memory and relation sizes of Experiment 2 (the moderate disk contention case) scaled up by a factor of 10, and with the arrival rates reduced by the same factor in order to maintain the resource utilizations at their previous levels.

In the case of the Max algorithm, which admits queries only with their maximum memory allocations, these changes should have no impact on the miss ratios since the maximum allocation of each query, $F\|R\|$, is unchanged relative to the memory size. In contrast to Max, the MinMax algorithm, which gives some queries their maximum required memory and others their minimum, would be affected by the larger sizes. This is because the minimum required memory is only $\sqrt{10}$ times larger than before, while the average query's maximum required memory and the system memory have both been increased by a factor of 10. Admitting extra queries with their minimum allocations would thus have a lesser impact on the memory allocations of high-priority queries. Consequently, the detrimental effect of MinMax would be reduced considerably, leading MinMax to deliver miss ratios that are much closer to those of the optimal MinMax-N algorithm. However, as we increase the arrival rate, the disadvantage of MinMax will still eventually overwhelm its benefits. For this reason, there is still a need for a mechanism to regulate query admissions.

Turning our attention to PMM, we first observe that PMM will still decide against using Max, as the behavior of the Max strategy is not affected by the larger sizes. Once in MinMax mode, the length of the MPL searching period should be about the same as before as a proportion of the (longer) query response times. This is because PMM will require roughly the same number of query completions to find the right MPL setting. Therefore, the qualitative behavior of PMM should remain the same as in Experiment 2. To verify this, we carried out two different sets of experiments — a set of medium-scale experiments, reported in this chapter, and a set of small-scale experiments that involved database and memory sizes that were ten times smaller. The two sets of experiments produced essentially the same qualitative algorithm behavior. We therefore expect our results to scale up to even larger memory and relation sizes; PMM should be just as effective for larger systems as it was for the workloads and configurations that we have experimented with here.

5.4. Conclusion

In this chapter, we have focused on the problem of scheduling queries in firm real-time database systems (RTDBS). As a solution to this problem, we have 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, which is usually not available in a database system. 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.

Using the detailed RTDBS simulation model described in Chapter 2, we studied the performance of PMM under workloads that comprised both hash joins and external sorts. For comparison purposes, we also examined two static algorithms based purely on the Max and MinMax allocation strategies. Our experiments revealed that while the static algorithms perform satisfactorily under very light loads, neither algorithm is adequate in overload situations. In contrast, PMM is able to dynamically reach the right compromise between Max and MinMax, consistently delivering low miss ratios. Moreover, PMM achieves this quickly enough so that it works well even for fluctuating workloads. While we only experimented with queries that perform either external sorting or hash join operations, PMM is designed to schedule general query workloads effectively by balancing their demands on the system's memory, CPU, and disks. In particular, PMM 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 the admission control and memory allocation mechanisms of PMM should be very useful for RTDBS query scheduling. However, when presented with a multiclass workload, PMM tends to produce skewed class miss ratios that may not be acceptable for certain applications. In the next chapter, we will investigate mechanisms that can help PMM to achieve class miss ratios that conform to administratively defined workload objectives.

CHAPTER 6

MULTICLASS QUERY SCHEDULING

As demonstrated in the previous chapter, the *Priority Memory Management* (PMM) algorithm is very effective in reaching appropriate multiprogramming levels (MPL) and memory allocation policies to minimize the system miss ratio. PMM is also capable of handling workload mixes that consist of classes with different average sizes, but it tends to favor some classes of queries while discriminating against other classes in such cases. Such a bias may not always be acceptable; for some applications, it may be desirable to distribute missed deadlines proportionally among all classes according to administratively-defined workload objectives. In order to achieve such controlled performance, the query scheduler must intervene on behalf of classes that, either because of stricter timing requirements or larger resource demands, are in a disadvantaged position if allowed to compete unaided for resources (e.g., see [Pang92]). Since resource allocation decisions are priority-driven, the most effective way to help disadvantaged classes is to boost their priorities relative to the advantaged classes. PMM therefore needs to be equipped with a mechanism that allows it to elevate or demote the priority of a query based on the observed relative performance of the class that it belongs to.

In this chapter, we augment the PMM algorithm with a class-priority adaptation mechanism, producing a *Priority Adaptation Query Scheduling* (PAQS) algorithm that is intended for handling multiclass query workloads where the natural biases of PMM are not acceptable. PAQS relies on PMM to set a system-wide MPL and a global memory allocation strategy; it then regulates the MPL and memory allocation of individual classes indirectly by controlling the priority of their queries. Roughly speaking, PAQS accomplishes this regulation using a multi-class variant of the Adaptive Earliest Deadline scheduling policy proposed in [Hari91]. PAQS 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 [Liu73], 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, PAQS is able to distribute missed deadlines among the query classes according to the specified workload objectives.

6.1. Priority Query Scheduling

This section presents two algorithms, PM3 and PAQS, for scheduling multiclass query workloads. The first algorithm, PM3, modifies the PMM algorithm (introduced in the previous chapter) to choose a multiprogramming level and a memory allocation strategy based on the multiclass workload objective. PM3 is likely to be adequate for some workload objectives, but it will not be sufficient for objectives that are more demanding since it lacks a mechanism to control the performance of individual classes. The PAQS algorithm, which augments PM3 with a class-priority adaptation mechanism, is intended to bridge this gap. Both algorithms accept as input a list of values, $RelMissRatio = \{relMissRatio_1 : \ldots : relMissRatio_{NumClasses}\}$, that states the desired miss ratio distribution among the classes in the workload. As an illustration, 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 $MissRatio_2 = x\%$. The details of the algorithms are presented below. The input parameters and the variables used by the two algorithms, which will be explained as they appear in the following description, are summarized in Table 6.1.

6.1.1. Priority Memory Management for Multiclass Workloads

As mentioned above, the *Priority Multiclass Memory Management* (PM3) algorithm is essentially an adaptation of PMM that is tailored to choose a system-wide target MPL and a global memory allocation strategy that are conducive to meeting the workload's multiclass performance objective. PM3 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.

Parameter	Meaning	Default
RelMissRatio	Target relative class miss ratios	{1 : ... : 1}
$SampleSize_{Class}$	Re-evaluation frequency (# completions per class)	10
$SampleSize_{Total}$	Re-evaluation frequency (total # completions)	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 computing weighted miss ratio	derived from *RelMissRatio*
P_Q	Priority of query *Q*	-
D_Q	Deadline of query *Q*	-
R_Q	Random key assigned to query *Q*	[0, 1]

Table 6.1: Notation for PM3 and PAQS

The primary mechanism that PM3 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 $relMissRatio_i = c \times relMissRatio_j$ 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 *RelMissRatio* into class weights:

$$Weight_i = \frac{\frac{1}{relMissRatio_i}}{\Sigma_j \frac{1}{relMissRatio_j}}$$

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

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

To illustrate how this procedure works, let us again consider a two-class workload with *RelMissRatio* = {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 PM3 chooses its memory allocation strategy. To adapt better in a multiclass context, PM3 needs to replace the system-wide performance measures that PMM uses with class-specific measures. PM3 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 the following conditions: (1) one or more queries from that class have missed their deadlines since PM3 was last activated; (2) class i has a non-zero 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, PM3 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, PM3 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 PM3's last activation.

Finally, to ensure that its choices of MPL setting and memory allocation strategy remain suitable for the workload, PM3 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, PM3 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 $Change_{ConfLevel}$ [Devo91]. If so, PM3 reacts to the workload change by discarding the statistics that it has gathered and by re-adapting itself to the new workload composition. The differences between PM3 and PMM are summarized in Table 6.2.

6.1.2. Priority Adaptation Query Scheduling

While PM3 is designed to pick its MPL and memory allocation strategy according to the target miss ratio distribution, it does not control the bias of the Earliest Deadline (ED) scheduling policy [Liu73] that RTDBSs use to prioritize their queries. As we will soon see, this bias can prevent PM3 from meeting its

	PMM	PM3
Miss ratio projection	$\text{AvgMissRatio} = \frac{\text{\# late queries}}{\text{\# queries}}$	WeightedMissRatio = $\Sigma\ \text{MissRatio}_i \times \text{Weight}_i$
Memory allocation	Switch from Max to MinMax if queries in the workload experience significant unnecessary memory waiting time	Switch from Max to MinMax if queries in some class experience significant unnecessary memory waiting time
Re-activation frequency	$\text{SampleSize}_{\text{Total}}$ query completions	(1) $\geq \text{SampleSize}_{\text{Total}}$ completions; and (2) $\geq \text{SampleSize}_{\text{Class}}$ completions/class
Restart condition	Changes in average workload characteristics	Changes in the characteristics of some class *i*

Table 6.2: Summary of Differences between PMM and PM3

given multiclass objective. To rectify this shortcoming, the *Priority Adaptation Query Scheduling* (PAQS) algorithm augments PM3 with a priority adaptation mechanism. This mechanism is intended to regulate the relative priority of individual classes, helping classes that would otherwise miss more deadlines to attain acceptable relative miss ratios.

As mentioned earlier, PAQS 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 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 delivers good performance under heavy loads [Hari91].

PAQS 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 the moment. At system start-up time, all *RegQuota*$_i$'s are first initialized to ∞ so that all queries are assigned to the regular group initially. When PAQS 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:

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

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, PAQS 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:

$$RegQuota_i = RegQuota_i \times \frac{Weight_i \times MissRatio_i}{WeightedMissRatio \, / \, NumClasses}$$

Returning to our *RelMissRatio* = {3 : 1} example, if currently the miss ratio of classes 1 and 2 are 20% and 10%, respectively, PAQS 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, PAQS continues to monitor the relative performance of the classes, applying the above formula to dynamically adapt the *RegQuota*$_i$'s as needed.

Admission control for PAQS is straightforward once the MPL has been determined — the *m* highest-priority queries get admitted, where *m* is the target MPL. However, the use of PAQS' two-tier priority scheme introduces 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 PAQS follows the same concept as the Adaptive Earliest Deadline (AED) algorithm proposed in [Hari91]. AED maintains a "hit" group and a "miss" group, which correspond to the regular group and reserve group in PAQS, 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 AED, the single *HitSize* parameter serves to stabilize the overload performance of the ED policy, whereas PAQS uses its $RegQuota_i$ values to influence the relative class miss ratios. Consequently, the procedures that the algorithms employ to set their control parameters are quite different.

6.2. Experiments and Results

This section presents the results of a series of experiments designed to evaluate the performance of the Priority Adaptation Query Scheduling (PAQS) algorithm and the Priority Multiclass Memory Management (PM3) 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 PAQS' effectiveness in achieving targeted relative class performances, and also to reveal any price (in terms of system-wide performance metrics) that PAQS may have to pay in the process. The performance of PM3 serves to illustrate the relative effectiveness of the priority adaptation mechanism that PAQS employs in attempting to meet its targets. As always, we will begin with a baseline experiment, with further experiments being carried out by varying a few parameters each time. The primary performance metrics for these experiments are the *class* miss ratio, *weighted* miss ratio, and *system* miss ratio (defined in Section 2.2).

6.2.1. Baseline Experiment

We begin our investigation of the performance of PAQS and PM3 with the multiclass workload that was used in the last experiment of the previous chapter. The detailed database and workload characteristics are repeated in Table 6.3 for ease of reference. The workload consists of two classes of hash joins, Medium and Small. Each join in the Medium class has two operand relations, **R** and **S**, where $\|R\|$ varies uniformly between 600 and 1800 pages and $\|S\|$ is selected from the range [3000, 9000] pages. Moreover, the slack ratio interval of this class is set to [2.5, 7.5], and its arrival rate is fixed at 0.065 queries/second. 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/second. The performance objective here is to **balance** the miss ratio of the two classes, i.e., *RelMissRatio* = {1 : 1}. 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; thus, memory is the bottleneck resource. This is achieved by letting *CPUSpeed* and *NumDisks* be 40 MIPS and 12, respectively, and by setting **M** to 2560 pages (20 MBytes). The rest of the RTDBS resource parameters are kept at their settings of Table 3.3.

Figures 6.1 and 6.2 plot the class miss ratios and system miss ratios produced by PMM, PM3, and PAQS as a function of the arrival rate of the Small class. 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/second, 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, PM3 and PAQS come much closer to achieving the objective of balanced miss ratios, though at the expense of higher system miss ratios. In fact, PAQS exhibits virtually no skewed

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
RelPerDisk$_3$	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	vary 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

Table 6.3: Database and Workload Parameter Settings for Baseline Experiment

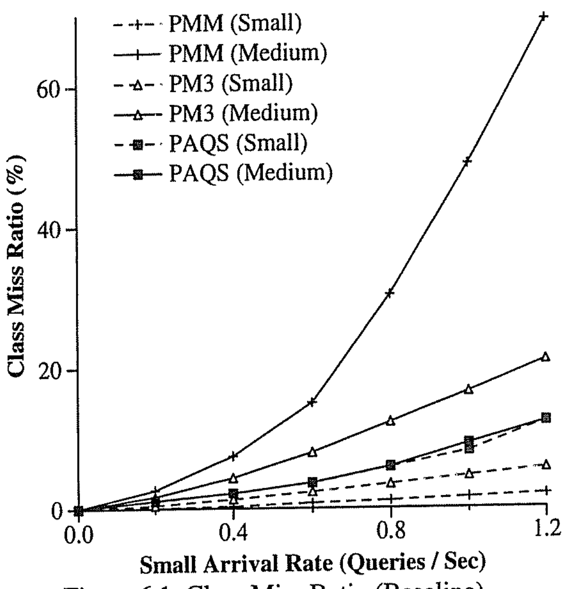

Figure 6.1: Class Miss Ratio (Baseline)

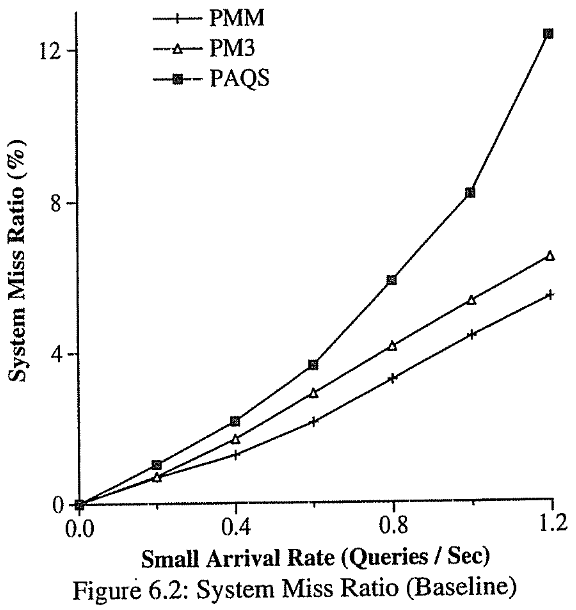

Figure 6.2: System Miss Ratio (Baseline)

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 three algorithms, we shall analyze each algorithm in turn with the aid of Figures 6.3 to 6.7, 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. 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 Small class, by virtue of its higher arrival rate, exerts a disproportional influence on the various measurements that PMM relies upon when making its target MPL and memory allocation strategy 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 exceeds 23 at a time (2560 memory pages divide by 111 pages per query). However, as the low observed MPLs in Figure 6.4 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

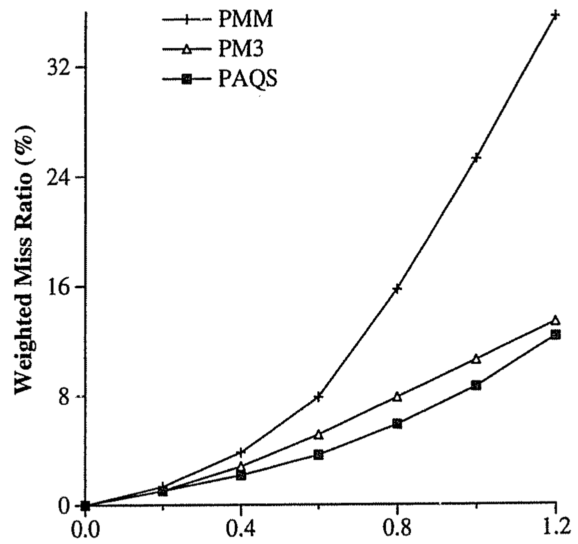

Figure 6.3: Weighted Miss Ratio (Baseline)

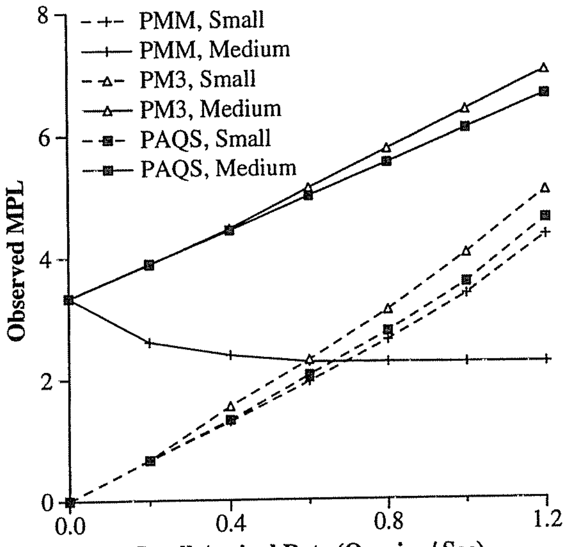

Figure 6.4: Observed MPL (Baseline)

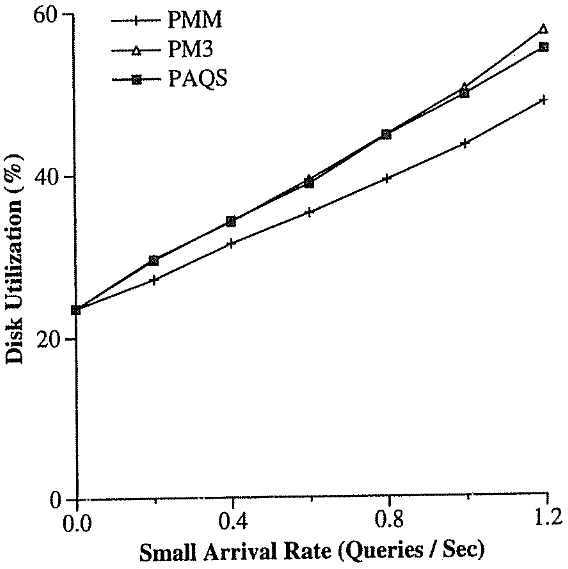

Figure 6.5: Disk Utilization (Baseline)

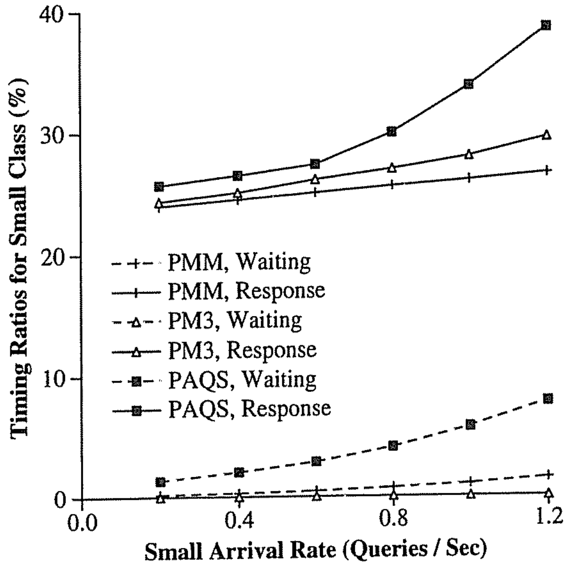

Figure 6.6: Small Timings (Baseline)

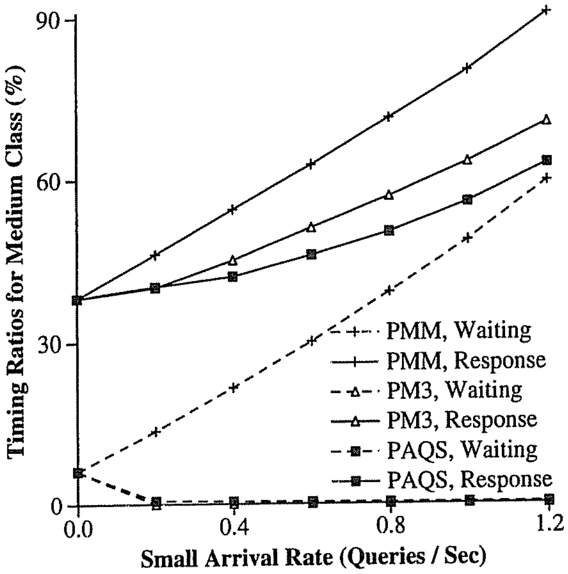

Figure 6.7: Medium Timings (Baseline)

get to execute concurrently, as each Medium join query's expected maximum memory requirement is 1321 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 Figure 6.5 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. As a result, Small queries are able to enjoy relatively short admission waiting and response times at the expense of the Medium class under PMM.

Another source of PMM's biased behavior is the Earliest Deadline policy used for resource scheduling [Pang92]. 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. Figures 6.6 and 6.7 provide evidence of this inherent bias in the ED policy. Even at a low load of $\lambda_{Small} = 0.2$ queries/second, Medium queries spend more than 10% of their deadlines waiting for admission and another 35% of their time constraints executing in the system (see Figure 6.7). In contrast, Small queries have negligible admission waiting times and finish way ahead of their deadlines (Figure 6.6). 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 $\lambda_{Small} = 1.2$ queries/second, 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. This bias in ED, together with PMM's biased MPL and memory allocation strategy choices, 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 PM3 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 in Figure 6.4 show that PM3 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 Figure 6.5 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 $\lambda_{Small} = 1.2$ queries/second to just over 20% for PM3. 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 PM3 delivers a more balanced miss ratio distribution and a much lower weighted miss ratio than PMM does. The higher system miss ratio that PM3 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 PM3 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, PM3 is able to arrive at the right MPL and memory allocation strategy.

Finally, we turn our attention to the PAQS algorithm. Besides possessing the MPL and memory allocation policy selection mechanisms of PM3, PAQS is also equipped with a priority adaptation mechanism to counteract any undesirable behavior due to the Earliest Deadline scheduling policy. Figure 6.8, which traces the percentage of queries in each class that are assigned to the PAQS reserve group, 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

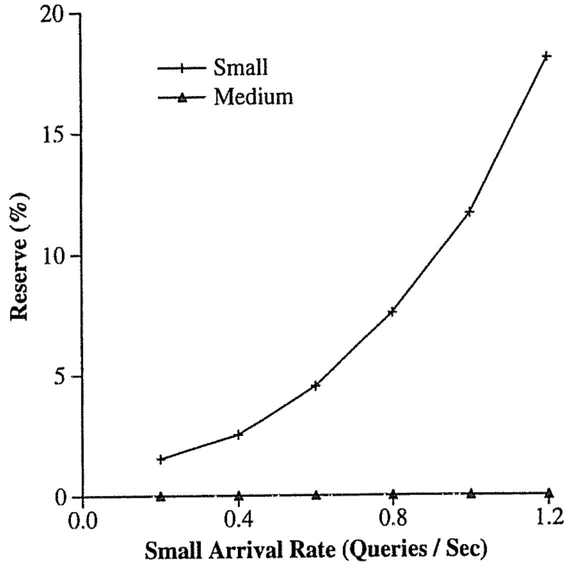

Figure 6.8: PAQS Reserve Group (Baseline)

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 under PM3 from the Medium class' higher concurrency) 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 PAQS' disk utilizations are lower than those of PM3 in Figure 6.5. This also helps Medium queries to execute faster and complete earlier, bringing their miss ratios down further to match those of the Small class. For example, at $\lambda_{Small} = 1.2$ queries/second, a Medium query requires an average of just over 60% of its time constraint to run under PAQS, whereas it takes more than 70% of its deadline when PM3 is employed. Consequently, PAQS is able to completely balance the class miss ratios, successfully meeting the workload objective. Interestingly, despite producing lower miss ratios for the Medium class than PM3, PAQS does not improve significantly upon PM3's weighted miss ratios. This is because PM3 already allows the system resources to be utilized productively, so PAQS 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, PM3 can come considerably closer to achieving balanced class miss ratios than PMM. Finally, by also manipulating the individual class quotas for regular queries, PAQS is able to influence their relative miss ratios enough to successfully eliminate any remaining shortcomings of PM3 in terms of accomplishing equitable miss ratios.

6.2.2. Skewed Class Objectives

Having demonstrated in the previous experiment that PAQS 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%. Figures 6.9 and 6.10 present the resulting class miss ratios and weighted miss ratios, while Figure 6.11 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 PM3 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 PM3, its behavior remains essentially unchanged because, even for the {2 : 1} target miss ratio distribution, it still misses more Medium queries than desired. Consequently, PM3 is already operating in a region where it is using the MPL setting and the memory allocation strategy that are most favorable to the Medium class, as it was in the previous experiment. We now examine PAQS. Not surprisingly, this 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 Figures 6.12 to 6.14. These figures show that while selecting the appropriate target MPL and memory allocation strategy almost enabled PM3 to meet the target miss ratio distribution of *RelMissRatio* = {2 : 1} earlier, PM3 is not able to improve the relative miss ratio of the Medium class any further when the workload objective necessitates it. PM3 fails miserably here, producing $MissRatio_{Small}$ / $MissRatio_{Medium}$ values that are far short of the target. In contrast, PAQS 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, PAQS still manages to bring the miss ratios of the Medium class down to meet the demanding workload objective. However, PAQS produces only slightly lower weighted miss ratios than PM3 here (Figure 6.13). As discussed in the baseline experiment, this is because the resource consumption of

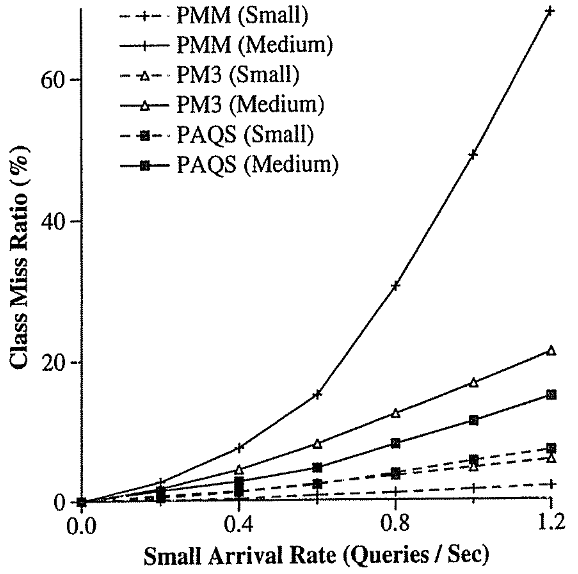

Figure 6.9: Class Miss Ratio (1:2)

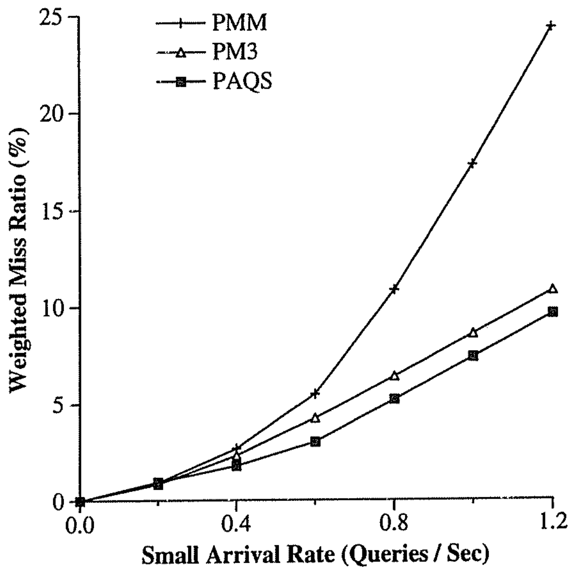

Figure 6.10: Weighted Miss Ratio (1:2)

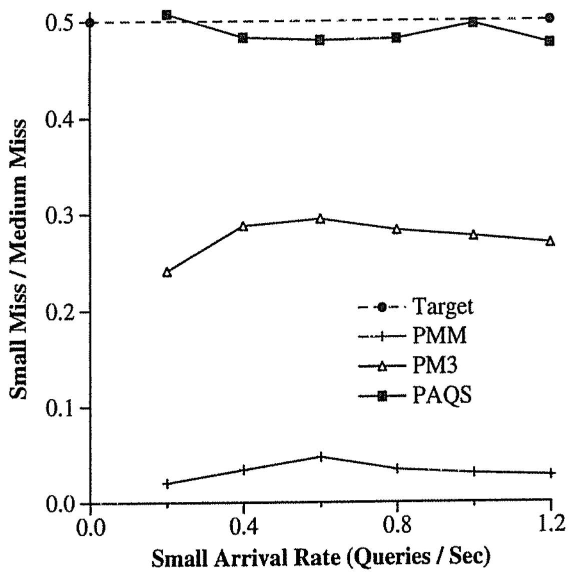

Figure 6.11: Class Miss Ratio Dist. (1:2)

Medium queries is much more than that of the Small queries, so the system has to sacrifice many more Small queries to achieve a reduction in the number of late Medium queries.

To summarize, the results of this experiment confirm that neither PMM nor PM3 are capable of achieving the target miss ratio distribution of multiclass workloads. In contrast, PAQS has demonstrated the ability

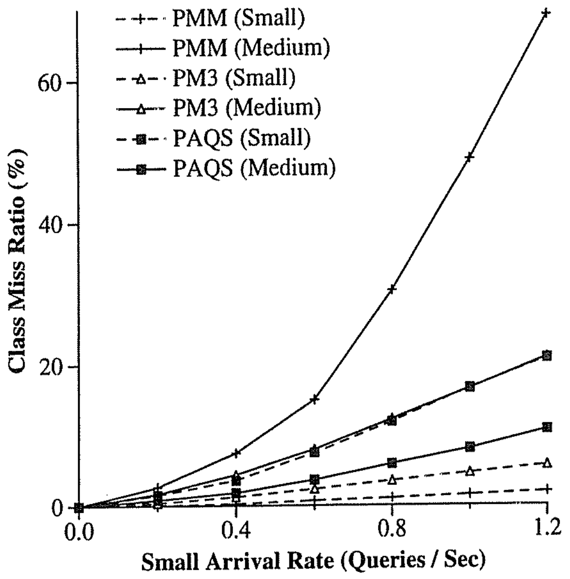

Figure 6.12: Class Miss Ratio (2:1)

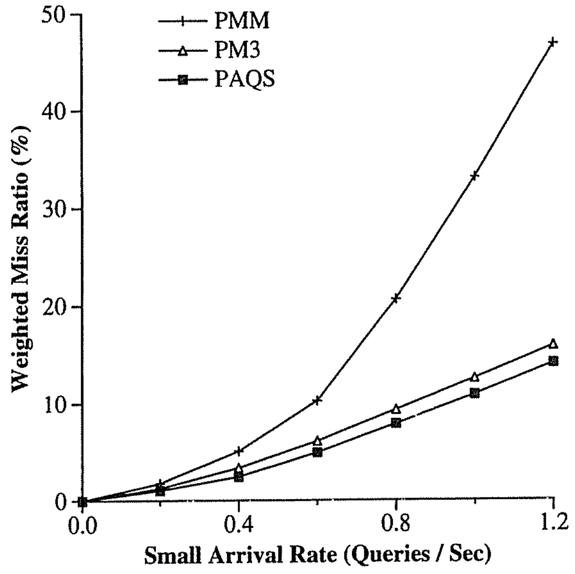

Figure 6.13: Weighted Miss Ratio (2:1)

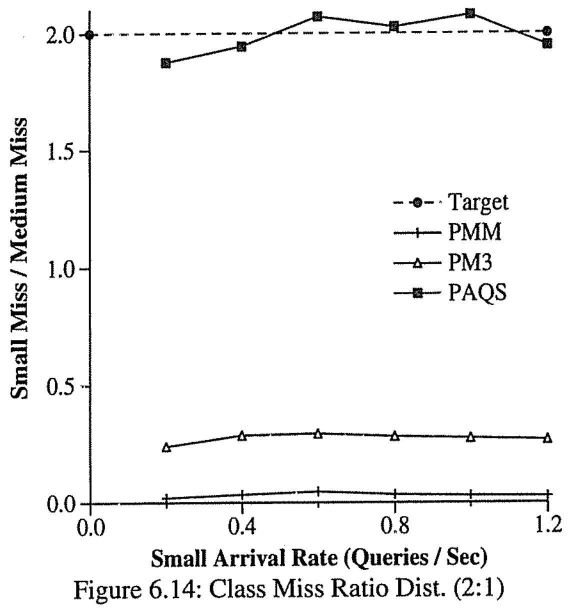

Figure 6.14: Class Miss Ratio Dist. (2:1)

to consistently meet multiclass performance objectives, whether balanced or skewed.

6.2.3. Identical Classes

In the first two experiments, we saw that the priority adaptation mechanism of PAQS is very effective in regulating per-class performance to achieve a desired target miss ratio distribution, even despite the classes' very different characteristics. However, the priority adaptation mechanism of PAQS 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.

Figure 6.15 plots the system miss ratios produced by the three query scheduling algorithms as a function of λ_{Small}. This figure shows that the performance of PMM and PM3 is, for all practical purposes, identical. The reason for this behavior is that PMM and PM3 both adopt the MinMax strategy to alleviate the memory bottleneck after detecting the memory contention experienced by queries in the two classes. Moreover, since both classes have identical characteristics, the system miss ratios that PMM uses and the weighted miss ratios that PM3 depends on yield very similar values, thus leading both algorithms to choose target MPLs that are almost the same. In contrast to PM3, PAQS produces slightly higher system miss ratios than PMM, indicating that the priority adaptation mechanism of PAQS indeed becomes a slight liability here. This

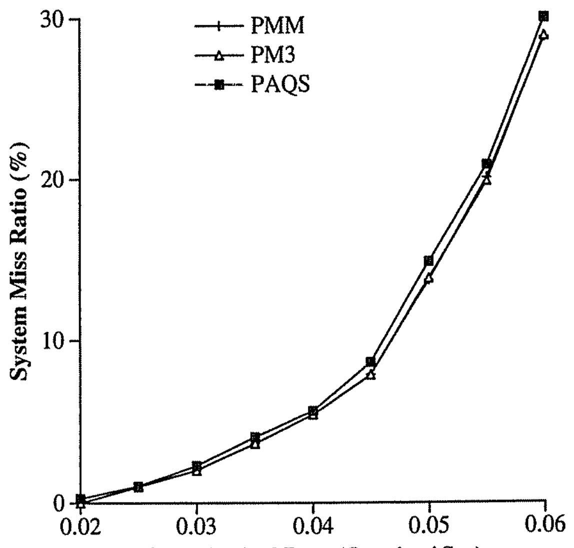

Figure 6.15: System Miss Ratio (Identical)

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, PAQS 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, PAQS suffers only a slight performance deterioration as a result. For example, at arrival rates of 0.06 queries/second, where both classes are missing as many as 29% of their queries under PMM and PM3, PAQS misses just about 30% of the queries. Consequently, while activating the priority adaptation mechanism unnecessarily can lead to some small overhead, PAQS's benefit of ensuring that the target miss ratio distribution is met appears to more than justify its use.

6.2.4. Workload Changes

The preceding experiments lead us to the conclusion that PAQS is very effective for relatively stable real-time workloads. The objective of this next experiment is to find out how well PAQS reacts to workload changes. This is done by subjecting the various query scheduling algorithms to a workload whose composition changes every 2 to 5 simulated hours. At any given time, the workload consists of two of the following query classes — Small, Medium, and Sort. The Small and Medium classes are the same as in the first two experiments. The Sort class is made up of external sorts. Each query in the Sort class sorts a single relation **R**, where $\|R\|$ ranges from 600 to 1800 pages. Table 6.4 summarizes the database and workload parameters (except arrival rates). The class arrival rates vary from one workload mix to another. To highlight the performance trade-offs between the various 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 6.5, while the resource parameters are the same as in 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.

Database	Value	Workload	Value
NumGroups	4	$QueryType_{Medium}$	Hash join
$RelPerDisk_1$	3	$RelGroup_{Medium}$	$\{1, 2\}$
$SizeRange_1$	[600, 1800] pages	$SRInterval_{Medium}$	[2.5, 7.5]
$RelPerDisk_2$	3	$QueryType_{Small}$	Hash join
$SizeRange_2$	[3000, 9000] pages	$RelGroup_{Small}$	$\{3, 4\}$
$RelPerDisk_3$	3	$SRInterval_{Small}$	[2.5, 7.5]
$SizeRange_3$	[50, 150] pages	$QueryType_{Sort}$	External Sort
$RelPerDisk_4$	3	$RelGroup_{Sort}$	$\{1\}$
$SizeRange_4$	[250, 750] pages	$SRInterval_{Sort}$	[2.5, 7.5]

Table 6.4: Database and Workload Parameter Settings (Workload Changes)

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

Table 6.5: Class Arrival Rates in queries/second (Workload Changes)

Table 6.6 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, all three algorithms 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 trade-offs between the three algorithms remain the same. In particular, PAQS is still the only algorithm that achieves the target miss ratio distribution. Turning our attention to workload mixture #2, we first note that PMM again unfairly discriminates against the Sort 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

Workload	PMM Avg. Miss Ratio			PM3 Avg. Miss Ratio			PAQS Avg. Miss Ratio		
Mix	Small	Medium	Sort	Small	Medium	Sort	Small	Medium	Sort
1	1.6%	44.3%	-	4.7%	16.5%	-	9.1%	9.6%	-
2	1.3%	-	79.0%	3.3%	-	9.6%	7.4%	-	7.3%
3	-	11.4%	10.1%	-	11.4%	10.2%	-	10.9%	10.8%

Table 6.6: Average Class Miss Ratios (Workload Changes)

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 read in a 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 allocation strategy that PMM chooses. In contrast, PM3 chooses MinMax and is considerably less biased, while PAQS again manages to balance the class miss ratios. Finally, PMM and PM3 generate similar miss ratios for workload mixture #3, as both adopt the MinMax mode and high MPL settings to service the two memory-intensive classes. Their slightly skewed miss ratios are a result of ED's favoring the Sort queries, which are somewhat shorter than the Medium hash join queries. This biased behavior is rectified by the priority adaptation mechanism of PAQS. This experiment shows that PAQS not only performs well under stable workloads, but is also capable of adapting to workload changes.

6.2.5. Three-Class Workloads

Up to this point, we have examined the performance of PAQS using workloads that consisted of only two classes in order to simplify our discussions. However, PAQS is intended to be a general multiclass query scheduling algorithm, and is not limited to handling only simple workloads. To demonstrate that PAQS is capable of managing more complex workloads well, we conclude this chapter 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/second, 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 Figures 6.16 to 6.18. 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. As for PM3, its miss ratios are quite balanced initially. Unfortunately, as λ_{Small} increases, the ED policy again causes PM3 to produce skewed class performance, though the performance

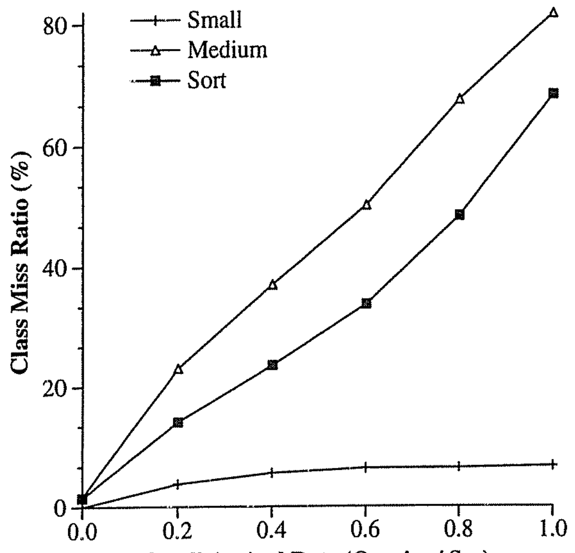

Figure 6.16: PMM (3 Classes)

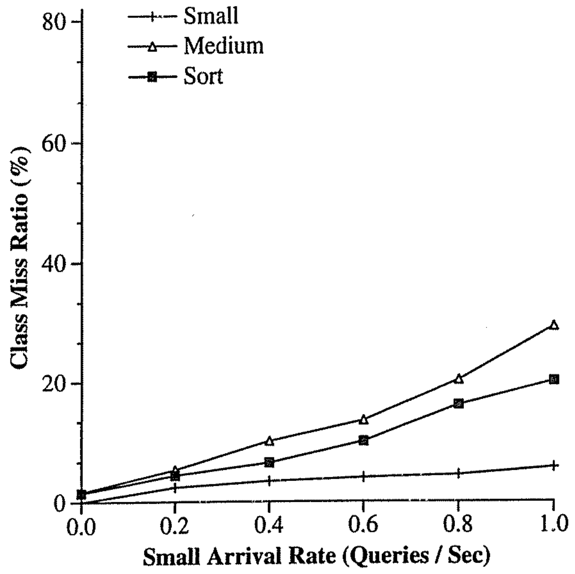

Figure 6.17: PM3 (3 Classes)

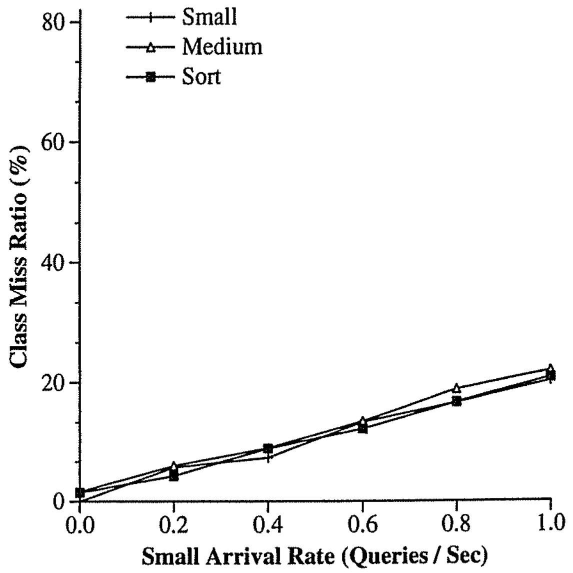

Figure 6.18: PAQS (3 Classes)

disparities between the classes are significantly less pronounced for PM3 than for PMM. Again, we see that PAQS is able to manipulate the priority of the classes appropriately to achieve the target miss ratio distribution.

6.3. Conclusion

In this chapter, we have addressed the problem of scheduling queries in firm real-time database systems (RTDBS) to meet multiclass performance objectives that are expressed in the form of target miss ratio distributions. As a solution to this problem, we have introduced a *Priority Adaptation Query Scheduling* (PAQS) algorithm. PAQS modifies the multiprogramming level (MPL) and memory allocation strategy selection mechanisms of the Priority Memory Management algorithm to pick a global MPL setting and a system-wide memory allocation strategy that are conducive to achieving the given target miss ratio 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 PAQS manipulates the relative priority of individual classes simply by adjusting their regular query quotas. By appropriately setting these quotas, PAQS is able to influence the miss ratios of the classes to conform to the target distribution.

Using the RTDBS simulation model presented in Chapter 2, we studied the performance of PAQS under a variety of workloads. Our experiments demonstrated that the priority adaptation mechanism of PAQS is very effective in helping it to attain the desired miss ratio distribution. Moreover, the modified MPL and memory allocation strategy selection mechanisms of PAQS enable it to utilize the system resources efficiently to reduce the overall number of deadline misses. Finally, PAQS was shown to be able to adapt to the offered workload quickly enough so that it works well even when there are workload changes. Overall, our results indicate that PAQS should be very useful for scheduling complex query workloads in an RTDBS.

CHAPTER 7

CONCLUSIONS

This thesis has focused on the problem of processing queries in *firm* real-time database systems (RTDBS), where jobs lose all value once their deadlines expire. The primary performance objective in an RTDBS is to minimize the number of missed deadlines. Due to the demanding nature of this objective, traditional first-come-first-serve or round-robin resource scheduling policies are inadequate for such systems. Instead, the resource schedulers of an RTDBS have to be priority-driven to ensure that queries receive their required resources in time to meet their deadlines. The adoption of priority scheduling in a database management system requires several changes in the ways that queries are processed. First, to facilitate query processing in an RTDBS, algorithms must be developed for query operators that allow queries to gracefully adjust to priority-induced reductions and increases in their memory allocations. Given such memory-adaptive algorithms, higher-level query scheduling algorithms must then be designed to efficiently handle the tasks of admission control, memory allocation, and priority assignment. These are the challenges that have been addressed in this dissertation.

7.1. Memory-Adaptive Query Processing Results

In the first two research chapters of this thesis, we investigated issues related to query execution in situations where the amount of memory available to a query may be reduced or increased during its lifetime. Chapter 3 studied the memory fluctuation problem in the context of hash joins. Our study demonstrated that simple approaches that react to a reduction in a join's allocated memory by suspending the join altogether or by paging the hash table of the join into and out of the remaining memory will not produce acceptable performance. This finding led us to propose a family of memory-adaptive hash join algorithms, called *Partially Preemptible Hash Join* (PPHJ). 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. The different PPHJ variants were derived by combining the following three techniques: (1) avoiding contracting the partitions of a join until it runs out of memory; (2) expanding contracted partitions while the outer relation is being divided; and (3) spooling output pages according to the page access pattern of the join. Performance studies revealed that although techniques (1) and (3) yield some performance gains, the biggest improvements come from the second technique. Overall, our results showed PPHJ with expansion to be a very effective technique for dealing with memory fluctuations.

Having understood how to efficiently adapt hash joins to fluctuations in their memory allocations, we then turned to the same problem for external sorts. In Chapter 4, we introduced an external sorting algorithm that begins by using a variation of replacement selection to split the operand relation into sorted runs; this replacement selection variant employs block writes to reduce the cost of disk seeks. Next, the sorted runs are 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 sub-steps that fit within the remaining memory if memory reductions occur. 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. Our performance evaluation indicated that this algorithm allows external sorts to execute efficiently in the face of memory fluctuations. The chapter also demonstrated how these techniques can be extended to sort-merge joins in order to make them memory-adaptive as well.

7.2. Real-Time Query Scheduling Results

With the low-level query primitives in place, we then turned our attention to developing high-level query scheduling policies that use the system's resources productively to meet query deadlines. The issues that arise here include admission control, memory allocation, and priority assignment. The last two research chapters of this thesis were devoted to meeting these higher-level query scheduling challenges.

In Chapter 5, 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 to its offered workload. 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. The choice of memory allocation strategy is based on statistics about the workload characteristics that PMM gathers. Performance studies using a wide range of workloads demonstrated that PMM is able to dynamically reach the right compromise between Max and MinMax, consistently delivering near-optimal miss ratios. Moreover, PMM achieves this quickly enough so that it works well even for fluctuating workloads.

Finally, Chapter 6 extended PMM to a *Priority Adaptation Query Scheduling* (PAQS) algorithm that is designed to meet multiclass performance objectives that are expressed in the form of target miss ratio distributions. PAQS divides the queries in an RTDBS into two priority groups — a regular group and a reserve group — and sets a quota of regular queries for each class. All regular queries are assigned higher priorities than any reserve query, allowing PAQS to manipulate the relative priority of individual classes simply by adjusting their regular query quotas. The PAQS algorithm was demonstrated to be capable of appropriately setting these quotas to meet the objectives of complex query workloads.

7.3. Future Work

A number of open issues remain in the area of real-time query scheduling. To begin with, there are at least two distinct avenues for further research on real-time query processing techniques: The first avenue is to explore additional strategies for adapting queries to memory fluctuations. One possible strategy would be to dynamically adjust the buffer size (i.e., the I/O block size) according to memory availability. Since the use of larger buffer sizes can lead to significant reductions in queries' response times [Haas93], a combination of buffer size adjustment and our proposed memory-adaptive techniques would likely yield even more effective solutions to the memory fluctuation problem. Another avenue for future work on query processing techniques is to introduce intra-query parallelism. While intra-query parallelism is not likely to be beneficial for systems that operate under heavy loads [Care88], it can be extremely useful in harnessing an RTDBS's resources under light load conditions. In fact, parallel query execution may be the only feasible alternative in situations where an RTDBS is presented with queries that have very tight deadlines. For this reason, it is important to investigate how an RTDBS should decide on the number of parallel processes to create for a given query, based on its operand relation sizes, its deadline, the system load, and so on. A related problem is that of physical data placement in RTDBSs. In a multi-disk RTDBS, relations can be declustered [Ries78, Livn87] to exploit the I/O bandwidth of the disks and to achieve load balancing. Issues that need to be addressed in declustering a relation include the choice of the number of disks on which to decluster the relation, i.e., the degree of declustering, and the choice of which particular disks to participate in the declustering. These choices need to take into account the timing requirements of individual classes in the workload. In addition, the decisions should consider the effectiveness of declustering in improving overall system performance.

There are several interesting possibilities at the query scheduling level as well. We have considered only workloads involving mixes of queries in this thesis; RTDBS workloads are likely to contain transactions as well as queries. Thus, it would be useful to combine long-term data buffering techniques, such as those proposed in [Brow93], with PAQS 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. Finally, PAQS typically takes 5 to 6 iterations to decide on the best MPL setting and memory allocation strategy (see Figures 5.5 and 5.14 for examples); it may be possible to shorten the adjustment time of the PAQS algorithm further by incorporating more sophisticated MPL control and memory allocation heuristics.

REFERENCES

[Abbo88a] R. Abbott, H. Garcia-Molina, "Scheduling Real-Time Transactions", *ACM SIGMOD Record, Vol. 17, No. 1*, March 1988.

[Abbo88b] R. Abbott, H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *Proc. of the 14th Int. Conf. on Very Large Data Bases*, August 1988.

[Abbo89] R. Abbott, H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data", *Proc. of the 15th Int. Conf. on Very Large Data Bases*, August 1989.

[Abbo90] R. Abbott, H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation", *Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS)*, December 1990.

[Baru91] S. Baruah, L. Rosier, "Limitations Concerning On-Line Scheduling Algorithms for Overloaded Real-Time Systems", *Proc. of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[Bitt88] D. Bitton, J. Gray, "Disk Shadowing", *Proc. of the 14th Int. Conf. on Very Large Data Bases*, August 1989.

[Blas77] M.W. Blasgen, K.P. Eswaran, "Storage and Access in Relational Databases", *IBM Systems Journal, Vol. 16, No. 4*, 1977.

[Brat84] K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations", *Proc. of the 10th Int. Conf. on Very Large Data Bases*, August 1984.

[Brow93] K.P. Brown, M.J. Carey, M. Livny, "Managing Memory to Meet Multiclass Workload Response Time Goals", *Proc. of the 19th Int. Conf. on Very Large Data Bases*, August 1993.

[Care88] M.J. Carey, M. Livny, "Parallelism and Concurrency Control Performance in Distributed Database Machines", *Proc. of the 14th Int. Conf. on Very Large Data Bases*, August 1988.

[Care89] M.J. Carey, R. Jauhari, M. Livny, "Priority in DBMS Resource Scheduling", *Proc. of the 15th Int. Conf. on Very Large Data Bases*, August 1989.

[Chen91] S. Chen, J.A. Stankovic, J.F. Kurose, D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems", *The Journal of Real-Time Systems, Vol. 3, No. 3*, September 1991.

[Corn89] D. Cornell, P. Yu, "Integration of Buffer Management and Query Optimization in a Relational Database Environment", *Proc. of the 15th Int. Conf. on Very Large Data Bases*, August 1989.

[Dert74] M. Dertouzos, "Control Robotics: the procedural control of physical processes", *Proc. of IFIP Congress*, 1974.

[Devo91] J.L. Devore, *Probability and Statistics for Engineering and the Sciences*, Brooks/Cole Pub. Co., 1991, pp. 283-301, 326-335.

[DeWi84] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M. Stonebraker, D. Wood, "Implementation Techniques for Main Memory Database Systems", *Proc. of the ACM SIGMOD Conf.*, June 1984.

[DeWi91] D.J. DeWitt, J.F. Naughton, D.A. Schneider, "Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting", *Proc. of the Int. Conf. on Parallel and Distributed Information Systems*, December 1991.

[Drap81] N.R. Draper, H. Smith, *Applied Regression Analysis, 2nd Edition*, John Wiley & Sons, Inc., 1981.

[Eswa76] K. Eswaran, et al, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of ACM*, November 1976.

[Grae90] G. Graefe, "Parallel External Sorting in Volcano", *Technical Report CU-CS-459-90*, University of Colorado, Boulder, March 1990.

[Grae91] G. Graefe, A. Linville, L.D. Shapiro, "Sort versus Hash Revisited", *Technical Report CU-CS-534-91*, University of Colorado, Boulder, July 1991.

[Gray79] J. Gray, "Notes on Database Operating Systems", in *Operating Systems: An Advanced Course*, R. Bayer, R.Graham, G. Seegmuller, eds., Springer-Verlag, 1979.

[Haas93] L. Haas, M.J. Carey, M. Livny, "SEEKing the Truth About *Ad Hoc* Join Costs", *Technical Report #1148*, Computer Sciences Department, University of Wisconsin - Madison, May 1993.

[Hari90a] J.R. Haritsa, M.J. Carey, M. Livny, "On Being Optimistic about Real-Time Constraints", *Proc. of the ACM PODS Symposium*, April 1990.

[Hari90b] J.R. Haritsa, M.J. Carey, M. Livny, "Dynamic Real-Time Optimistic Concurrency Control", *Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS)*, December 1990.

[Hari91] J.R. Haritsa, M. Livny, M.J. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems", *Proc. of the 12th IEEE Real-Time Systems Symposium (RTSS)*, December 1991.

[Hari92] J.R. Haritsa, M.J. Carey, M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems", *Real-Time Systems Journal, Vol. 4, No. 3*, September 1992.

[Hari93] J.R. Haritsa, M.J. Carey, M. Livny, "Value-Based Scheduling in Real-Time DBS", *VLDB Journal, Vol. 2, No. 2*, April 1993.

[Hong93] D. Hong, T. Johnson, S. Chakravarthy, "Real-Time Transaction Scheduling: A Cost Conscious Approach", *Proc. of the ACM SIGMOD Conf.*, May 1993.

[Huan89] J. Huang, J.A. Stankovic, D. Towsley, K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing", *Proc. of the 10th IEEE Real-Time Systems Symposium (RTSS)*, December 1989.

[Huan91] J. Huang, J.A. Stankovic, D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *Proc. of the 17th Int. Conf. on Very Large Data Bases*, September 1991.

[Jauh90] R. Jauhari, M.J. Carey, M. Livny, "Priority Hints: An Algorithm for Priority-Based Buffer Management", *Proc. of the 16th Int. Conf. on Very Large Data Bases*, August 1990.

[Jens85] E. Jensen, C. Locke, H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems", *Proc. of the 6th IEEE Real-Time Systems Symposium (RTSS)*, December 1985.

[Kim91] W. Kim, J. Srivastava, "Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling", *Proc. of the 12th IEEE Real-Time Systems Symposium (RTSS)*, December 1991.

[Kits83] M. Kitsuregawa, H. Tanaka, T. Moto-oka, "Application of Hash to Data Base Machine and Its Architecture", *New Generation Computing, Vol. 1, No. 1*, 1983.

[Kits89] M. Kitsuregawa, M. Nakayama, M. Takagi, "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method", *Proc. of the 15th Int. Conf. on Very Large Data Bases*, August 1989.

[Klei76] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, John Wiley & Sons, Inc., 1976, pp. 166-170.

[Knut73] D. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA., 1973.

[Kort90] H.F. Korth, N. Soparkar, A. Silberschatz, "Triggered Real-Time Databases with Consistency Constraints", *Proc. of the 16th Int. Conf. on Very Large Data Bases*, August 1990.

[Liu73] C. Liu, J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, January 1973.

[Livn87] M. Livny, S. Khoshafian, H. Boral, "Multi-Disk Management Algorithms", *Proc. of the ACM SIGMETRICS Conf.*, May 1987.

[Livn90] M. Livny, "DeNet User's Guide, Version 1.5", Computer Sciences Department, University of Wisconsin - Madison, 1990.

[Lock86] C. Locke, "Best Effort Decision Making for Real-Time Scheduling", *Ph.D. Thesis*, Department of Computer Science, Carnegie-Mellon University, May 1986.

[Mena82] D. Menasce, T. Nakanishi, "Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems", *Information Systems, Vol. 7, No. 1*, 1982.

[Mok78] A. Mok, M. Dertouzos, "Multi-processor Scheduling in a Hard Real-Time Environment", *Proc. of the 7th Texas Conf. on Computing Systems*, October 1978.

[Naka88] M. Nakayama, M. Kitsuregawa, M. Takagi, "Hash-Partitioned Join Method Using Dynamic Destaging Strategy", *Proc. of the 14th Int. Conf. on Very Large Data Bases*, August 1988.

[Pang92] H. Pang, M. Livny, M.J. Carey, "Transaction Scheduling in Multiclass Real-Time Database Systems", *Proc. of the 13th IEEE Real-Time Systems Symposium (RTSS)*, December 1992.

[Pang93a] H. Pang, M.J. Carey, M. Livny, "Partially Preemptible Hash Joins", *Proc. of the ACM SIGMOD Conf.*, May 1993.

[Pang93b] H. Pang, M.J. Carey, M. Livny, "Memory-Adaptive External Sorting", *Proc. of the 19th Int. Conf. on Very Large Data Bases*, August 1993.

[Pang94] H. Pang, M.J. Carey, M. Livny, "Managing Memory for Real-Time Queries", *Proc. of the ACM SIGMOD Conf.*, to appear, 1994.

[Panw88] S. Panwar, D. Towsley, "On the Optimality of the STE Rule for Multiple Server Queues that Serve Customers with Deadlines", *COINS Technical Report 88-81*, University of Massachusetts, Amherst, July 1988.

[Pete86] J.L. Peterson, A. Silberschatz, *Operation System Concepts*, Addison Wesley, 1986.

[Ries78] D. Ries, R. Epstein, "Evaluation of Distribution Criteria for Distributed Database Systems", *UCB/ERL Technical Report M78/22*, UC Berkeley, May 1978.

[Robi82] J. Robinson, "Design of Concurrency Controls for Transaction Processing Systems", *Ph.D. Thesis*, Carnegie Mellon University, 1982.

[RTS92] *Real-Time Systems, 4(3)*, Special Issue on Real-Time Databases, September 1992.

[Salz90] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, B. Vaughan, "FastSort: A Distributed Single-Input Single-Output External Sort", *Proc. of the ACM SIGMOD Conf.*, May 1990.

[Sarg76] R. Sargent, "Statistical Analysis of Simulation Output Data", *Proc. of the 4th Annual Symposium on the Simulation of Computer Systems*, August 1976.

[Sha90] L. Sha, R. Rajkumar, J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transactions on Computers, Vol. 39, No. 9*, September 1990.

[Shap86] L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems, Vol. 11, No. 3*, September 1986.

[SIGM88] *ACM SIGMOD Record, Vol. 17, No. 1*, Special Issue on Real-Time Data Base Systems, S. Son, editor, March 1988.

[Stan88] J.A. Stankovic, W. Zhao, "On Real-Time Transactions", *ACM SIGMOD Record, Vol. 17, No. 1*, March 1988.

[Ston81] M. Stonebraker, "Operating System Support for Database Management", *Comm. of the ACM, Vol. 24, No. 7*, 1981.

[Yu93] P.S. Yu, D.W. Cornell, "Buffer Management Based on Return on Consumption In a Multi-Query Environment", *VLDB Journal, Vol. 2, No. 1*, January 1993.

[Zell90] H. Zeller, J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments", *Proc. of the 16th Int. Conf. on Very Large Data Bases*, August 1990.