Partially Preemptible Hash Joins

HweeHwa Pang
Michael J. Carey
Miron Livny

Technical Report #1144

April 1993

Partially Preemptible Hash Joins

HweeHwa Pang
Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin - Madison

Abstract

With the advent of real-time and goal-oriented database systems, priority scheduling is likely to be an important feature in future database management systems. A consequence of priority scheduling is that a transaction may lose its buffers to higher-priority transactions, and may be given additional memory when higher-priority transactions leave the system. Due to their heavy reliance on main memory, hash joins are especially vulnerable to fluctuations in memory availability. Previous studies have proposed modifications to the hash join algorithm to cope with these fluctuations, but the proposed algorithms have not been extensively evaluated or compared with each other. This paper contains a performance study of these algorithms. In addition, we introduce a family of memory-adaptive hash join algorithms that turns out to offer even better solutions to the memory fluctuation problem that hash joins experience.

This work was partially supported by a scholarship from the Institute of Systems Science, National University of Singapore, and by an IBM Research Initiation Grant.

An abridged version of this paper appears in the proceedings of the ACM SIGMOD Conference, May 1993

1. Introduction

Database management systems (DBMS) are faced with increasingly demanding performance objectives. These objectives could be time constraint requirements, as in real-time database systems [SIGM88, Abbo88, Huan89, Hari90, Kort90, Kim91, REAL92], or administration defined performance goals as in goal-oriented database systems [Ferg93]. Traditional first-come-first-serve or round-robin scheduling policies are no longer adequate to meet such objectives; a DBMS has to prioritize transactions that are competing for system resources according to the system objectives and the resource requirements of the transactions.

With priority scheduling, the DBMS may preempt a transaction that is currently allocated a resource when that resource is requested by a higher-priority transaction. To avoid severe performance degradation, e.g. due to convoys that arise when transactions holding critical resources are suspended [Blas77], it is desirable to preempt a transaction only at a preemption-safe point, where the transaction is not holding any critical resources and the preemption cost is minimal [Ston81]. Scans and updates, which acquire and release resources repeatedly throughout their lifetimes, have frequent preemption-safe points. In contrast, large joins, especially hash joins, hold on to their buffers for an extended period each time. The DBMS therefore cannot wait for these joins to reach their preemption-safe points before preempting them. When a join has to be preempted prematurely, measures have to be taken to minimize the performance penalty of preemption.

To execute efficiently, a hash join requires a significant amount of main memory to hold its hash table. 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 the inner relation to the inner relation size [DeWi84, Shap86], which can be a substantial portion of the system memory. Moreover, this hash table has to be kept in memory for a long period of time. Consequently, during the lifetime of a large hash join, the DBMS may have to appropriate some of the join's memory to satisfy the memory requirements of higher-priority transactions; the buffers that are taken away may subsequently be returned after those transactions leave the system. Given the prospect of continually having memory taken away and given back during its lifetime, a hash join has to adapt its buffer usage to minimize any detrimental effect that might result from the changes in its allocated memory. To simplify our discussion, we shall henceforth refer to these changes as memory fluctuations.

One way to deal with memory fluctuations would be for the DBMS to employ virtual memory techniques to page the hash table of an affected hash join into and out of a smaller region of allocated memory, without having to inform the join operator. If the DBMS detects that this is causing too many page faults, it could suspend the join altogether.

An advantage of this approach is that it shields the hash join algorithm from the complexity involved in adapting to memory fluctuations. However, there may be severe performance drawbacks associated with this approach. On one hand, suspending hash joins that are affected by memory fluctuations reduces the number of active transactions, which may lead to under-utilization of system resources. Paging the hash table of a join, on the other hand, is likely to result in thrashing; a hash join accesses its hash table pages randomly, and any page that is replaced will likely be needed before long.

In this study, we investigate a different approach, namely, to involve the affected hash joins in adapting to the fluctuations. These algorithms range from relatively simple ones, which require few extensions to the original hash join algorithm, to sophisticated algorithms that dynamically adjust the buffer usage of hash joins to reduce the performance penalty that results from memory fluctuations. The second group of algorithms includes a family of hash join variants called *Partially Preemptible Hash Join* (PPHJ). All the PPHJ variants are capable of dynamically adjusting the buffer usage of a join in reaction to a drop in the amount of memory allocated to it (hence partially preemptible), or an increase in the allocated memory. They differ from one another in the way that they prepare for the event of memory shortage, and in the way that they utilize excess memory. Together, these algorithms cover a wide range of choices in dealing with fluctuations in memory availability. To understand the performance trade-offs of each algorithm and to identify those algorithms that adapt well to changes in system buffer usage, we have constructed a detailed simulation model of a database system. This model enables us to study the behavior of the hash join algorithms over a wide range of system resource configurations.

The remainder of this paper is organized as follows. Section 2 reviews the GRACE and Hybrid Hash Join algorithms, two hash join algorithms that we will use, and discusses existing work that is related to our study. The family of PPHJ algorithms is introduced in Section 3. Also included in Section 3 is a description of the algorithms that will be studied along with PPHJ. A detailed simulator of a database system, intended for studying the performance of the various algorithms, is described in Section 4. Section 5 presents the results of a series of simulation experiments showing that, over a wide range of system conditions, PPHJ offers effective solutions to the problem of memory fluctuations. Finally, our conclusions are presented in Section 6.

2. Related Work

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 paper.

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 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 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 ıot 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 sets 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, so that the **R** and S tuples that belong to this partition can be joined in memory directly as S is being scanned. The Hybrid Hash Join algorithm was shown to have superior performance over GRACE.

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 1: Notation

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 misfit 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 introduction), 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. Memory-Adaptive Hash Join Algorithms

This section gives a detailed description of the memory-adaptive hash join algorithms that will be examined in this study. 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.

3.1. 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.

3.1.1. 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].

(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.)

3.1.2. 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] In the experiments that are reported in this paper, 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 [Pang93], 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.

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. Other Algorithms

3.2.1. 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 this section), 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. 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*).

3.2.3. GRACE and Hybrid

Besides PPHJ, NKT' and ZG', we also include the GRACE and Hybrid Hash Join algorithms in this 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.

4. Database System Simulation Model

To aid in our on-going research on real-time database systems, we have constructed a simulation model of a centralized database system. The portion of our simulation model that is relevant to this study is shown in Figure 1. There are five components: a *Source* that generates transactions one after another, and also collects statistics on completed transactions; a *Transaction Manager* that models the execution of transactions, including joins; a *Buffer Manager* that implements the buffer management policy; and a *CPU Manager* and a *Disk Manager* that are responsible for managing the system's CPU and disks, respectively. In this section, we describe how the simulation model captures the details of the database, workload, and various physical resources of a database system. The simulator is written in the DeNet

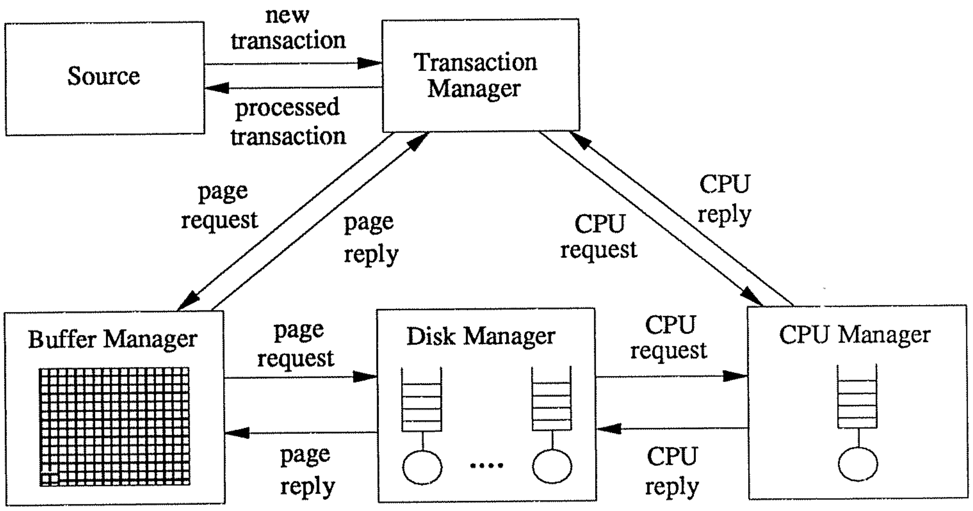

Figure 1: Database System Model

simulation language [Livn90].

4.1. Database and Workload Model

Table 2 summarizes the database and workload model parameters that are relevant to this study. Our objective is to simulate a stream of binary hash joins on different source relations. To facilitate this, the database consists of two groups of relations. There are $NumRel_1$ relations in the first group, and $NumRel_2$ relations in the second. Each relation i,j ($i = 1, 2$; $1 \le j \le NumRel_i$), in turn, has a size of $RelSize_{i,j}$ MBytes and occupies contiguous pages on disk. If there are multiple disks, all relations are declustered (horizontally partitioned) [Ries78, Livn87] across all of the disks. To minimize disk head movement, the relations are allotted the middle cylinders of the disks; temporary files occupy either the inner cylinders or the outer cylinders.

DB Model Parameters	Meaning
$NumRel_i$	Number of relations in group i
$RelSize_{i,j}$	Size of relation j of group i
$TupleSize_{i,j}$	Average tuple size of relation i,j in bytes
Workload Parameters	**Meaning**
$Duration_{MemReq}$	Mean duration of memory requests
MemReqThreshold	Maximum % buffer demand of a "small" memory request
Prob(SmallMemReq)	Probability of "small" memory request

Table 2: Database and Workload Model Parameters

In this study, the workload is made up of a series of joins; a new join is submitted to the database system only when the previous join has been completed. Each join involves an inner relation **R**, which is uniformly selected from the relations in the first group, and an outer relation **S** which is uniformly selected from the second group. 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.

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 "transactions" that have small memory requirements and, occasionally, with "transactions" that have large buffer demands. The memory contention experienced by the active joins is modelled here by a simple stream of competing 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 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.

4.2. Physical Resource Model

The parameters that specify the physical resources of our model, which consist of one CPU, multiple disks, and main memory, are listed in Table 3. There is a single queue for the CPU, and a first-come-first-serve (FCFS) scheduling discipline is used. The MIPS rating of the CPU is given by *CPUSpeed*. Table 4 gives the cost of various CPU operations that are involved in the execution of a join. These CPU costs are based on instruction counts taken from the Gamma database machine [DeWi90].

Turning to the disk model, *NumDisks* specifies the number of disks attached to the system. Each disk has its own queue and disk requests are serviced according to the elevator algorithm. The characteristics of the disks are also given in Table 3. Using the parameters in this table, the total time required to complete a disk access is computed as:

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

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

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

Parameter	Meaning	Parameter	Meaning
CPUSpeed	MIPS rating of CPU	*CylinderSize*	Number of pages per cylinder
NumDisks	Number of disks	*PageSize*	Number of bytes per page
SeekFactor	Seek factor of disk	**M**	Total system memory
RotationTime	Time for one disk rotation	*SleepTime*	Time between memory write activations
NumCylinders	Number of cylinders per disk	*FlushThreshold*	Threshold for memory writes

Table 3: Physical Resource Model Parameters

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

Table 4: Number of CPU Instructions Per Operation

Finally, the system has a total memory size of **M** MBytes. A memory reservation mechanism is provided to allow operators, including joins, to reserve buffers. Buffers that are reserved are managed by the operators themselves. Page replacement for non-reserved pages is handled as follows: The DBMS first attempts to find the least recently used clean page for replacement, avoiding the dirty pages initially. If there is no clean page, then the least recently used dirty page is selected. Before a dirty page can be replaced, however, its contents need to be written to disk. This lengthens the time that is needed to satisfy the waiting buffer request, and should be avoided if possible. For this reason, an asynchronous memory write process is provided to flush dirty pages to disk periodically [Teng84]. The write process is activated every *SleepTime* seconds. Upon activation, the process flushes all of the dirty pages that are older than *FlushThreshold.* The reason for flushing only the "old" dirty pages is to prevent unnecessary writes of pages that are frequently updated.

5. Experiments and Results

In this section, the database system simulator described in Section 4 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 5.

To ensure the statistical validity of our results, we verified that the 90% confidence intervals for join response times (computed using the batch means approach [Sarg76]) were sufficiently tight. The size of these confidence inter-

Indicator	Algorithm
PPHJ	Partially Preemptible Hash Join
early vs *late*	Early vs Late Contraction
noexp vs *exp*	No Expansion vs Expansion
lru vs *prio*	LRU vs 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 5: Algorithm Indicators

vals was within a few percent of the mean in almost all cases, which is more than sufficient for our purposes. Throughout the paper we discuss only statistically significant performance differences.

5.1. 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. Moreover, to model primary key-foreign key joins, we let $\|R\|$ and $\|S\|$ be 2 MBytes and 20 MBytes, respectively, and **M** be 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.) Finally, the memory write process is activated every second. Upon activation, the process flushes all of the dirty pages. Therefore *SleepTime* and *FlushThreshold* are 1 second and 0 seconds, respectively. The parameter settings for this experiment are summarized in Table 6.

Figure 2 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

DB Model Parameter	**Value**	**Resource Parameter**	**Value**
NumRel$_1$	5	*CPUSpeed*	20 MIPS
NumRel$_2$	5	*NumDisks*	1
RelSize$_{1j}$	2 MBytes	*SeekFactor*	0.000617
RelSize$_{2j}$	20 MBytes	*RotationTime*	16.7 msec
TupleSize$_{i,j}$	256 Bytes	*NumCylinders*	1500
		CylinderSize	90 pages
Workload Parameter	**Value**	*PageSize*	8 KBytes
		M	3.2 MBytes
		SleepTime	1 second
Duration$_{MemReq}$	1 second	*FlushThreshold*	0 second
MemReqThreshold	20%		
Prob(SmallMemReq)	0.8		

Table 6: Parameter Settings for the Baseline Experiment

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.

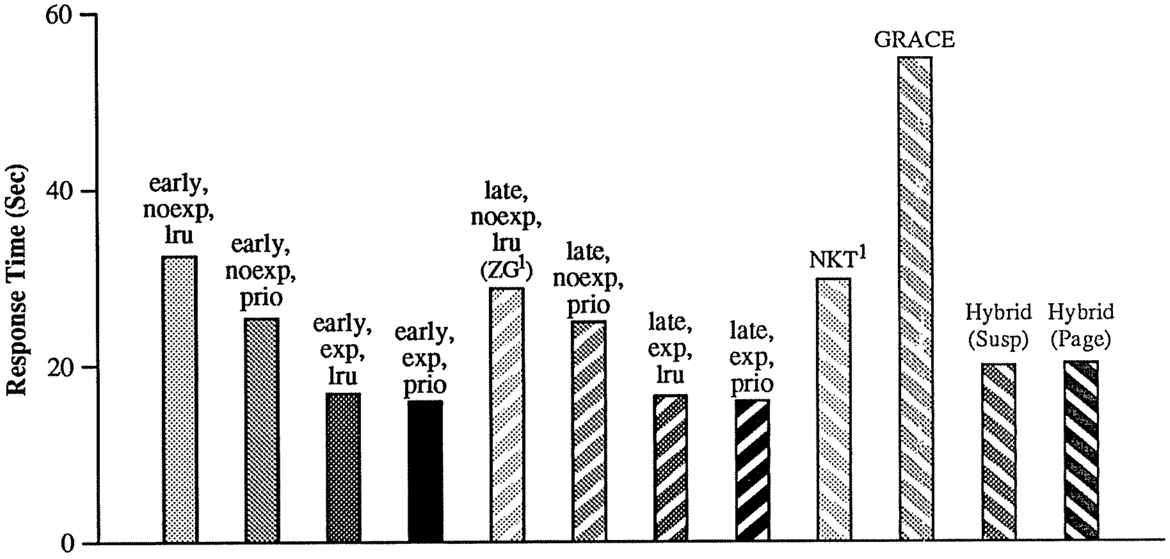

Figure 2: $||R||$ = 2 MBytes, $||S||$ = 20 MBytes, **M** = 3.2 MBytes

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 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 7, 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 7 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 7 again. With expansion, the number of S-I/Os is 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 7 into Table 8 to highlight the relative contributions of LRU spooling versus priority spooling. Table 8 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

	No Expansion (*noexp*)				**Expansion (*exp*)**			
	R-I/O	**S-I/O**	**Overhead-I/O**	**Resp.**	**R-I/O**	**S-I/O**	**Overhead-I/O**	**Resp.**
early,lru	310	2001	2311	32.4	322	77	399	16.8
early,prio	290	1724	2014	25.4	302	77	379	15.9
late,lru	304	1790	2094	28.8	310	72	382	16.5
late,prio	275	1675	1950	24.9	304	75	379	15.9

Table 7: Expansion Mechanism

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 9 focuses on the impact of the timing of contraction. Late contraction consistently produces lower R-I/Os and S-I/Os 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.

	LRU Spooling (*lru*)				**Priority Spooling** (*prio*)			
	R-I/O	**S-I/O**	**Overhead-I/O**	**Resp.**	**R-I/O**	**S-I/O**	**Overhead-I/O**	**Resp.**
early,noexp	310	2001	2311	32.4	290	1724	2014	25.4
early,exp	322	77	399	16.8	302	77	379	15.9
late,noexp	304	1790	2094	28.8	275	1675	1950	24.9
late,exp	310	72	382	16.5	304	75	379	15.9

Table 8: LRU vs Priority Spooling

	Early Contraction (*early*)				**Late Contraction** (*late*)			
	R-I/O	**S-I/O**	**Overhead-I/O**	**Resp.**	**R-I/O**	**S-I/O**	**Overhead-I/O**	**Resp.**
noexp,lru	310	2001	2311	32.4	304	1790	2094	28.8
noexp,prio	290	1724	2014	25.4	275	1675	1950	24.9
exp,lru	322	77	399	16.8	310	72	382	16.5
exp,prio	302	77	379	15.9	304	75	379	15.9

Table 9: Early vs Late Contraction

Turning to NKT' in Figure 2, 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 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.

5.2. 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 Table 6. Figure 3 gives the performance results. We will focus only on behaviors that differ significantly from those observed in the previous experiment.

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 10. 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.

Turning our attention to spooling (*lru* vs. *prio*) in Figure 3, we again see that priority spooling produces only a slight performance improvement over LRU spooling. In this experiment, where memory shortages occur frequently,

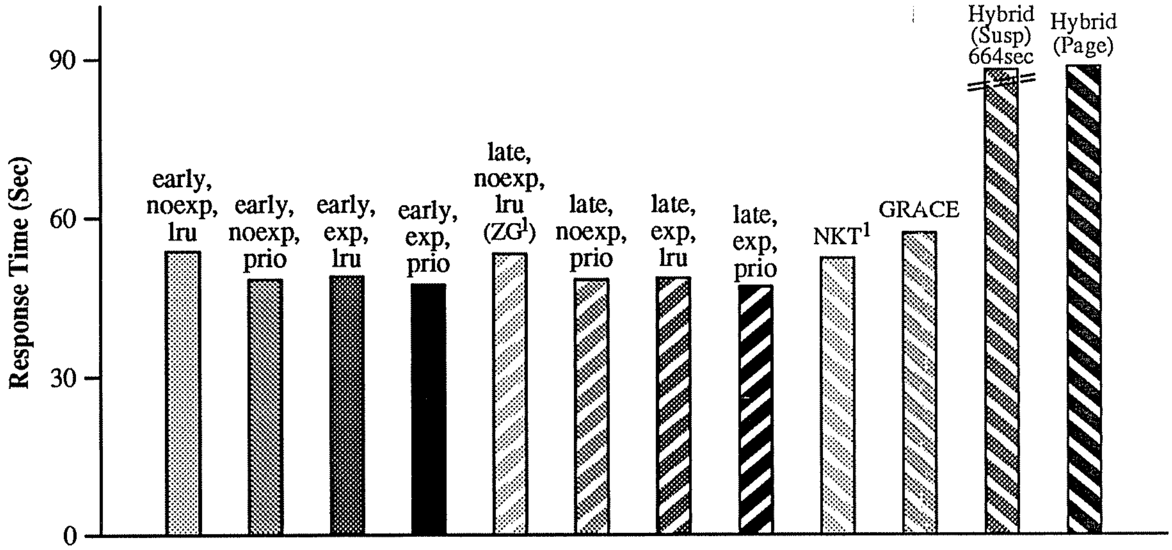

Figure 3: ||*R*|| = 2 MBytes, ||*S*|| = 20 MBytes, M = 0.8 MBytes

	No Expansion (*noexp*)				Expansion (*exp*)			
	R-I/O	**S-I/O**	**Overhead-I/O**	**Resp.**	**R-I/O**	**S-I/O**	**Overhead-I/O**	**Resp.**
early,lru	473	4571	5044	53.7	897	3367	4264	49.0
early,prio	472	4570	5042	48.4	816	3360	4176	47.4
late,lru	472	4549	5021	53.3	887	3306	4193	48.7
late,prio	471	4522	4993	48.4	796	3310	4106	47.1

Table 10: Expansion Mechanism

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 shows that it is only 15% 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.

5.3. 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 Table 6. 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 4 plots the response time of the five algorithms against **M**. This figure shows that no algorithm clearly dominates 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 5 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 5. Moreover, the curves for PPHJ(*late,exp,prio*) and PPHJ(*early,exp,lru*) are almost the same, which lends 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

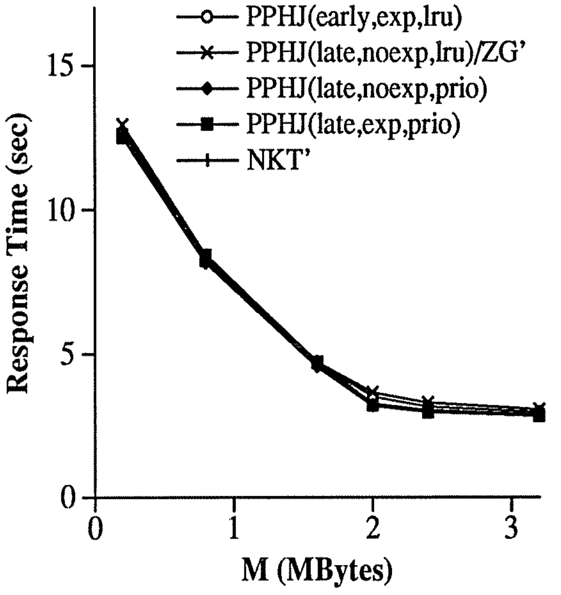

Figure 4: ||*R*|| = ||*S*|| = 2 MBytes

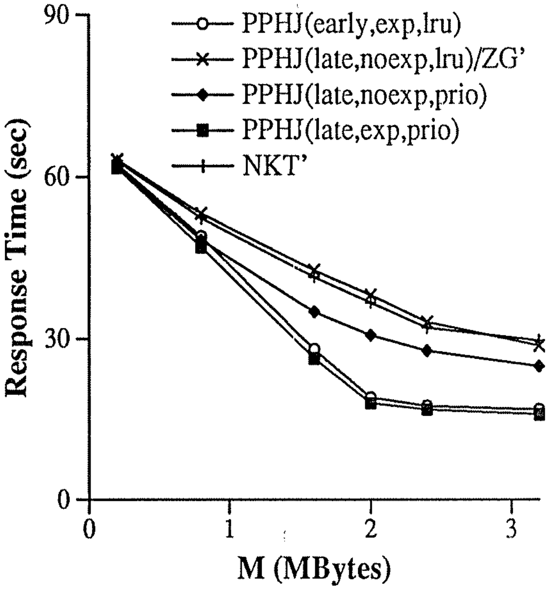

Figure 5: ||*R*|| = 2 MBytes, ||*S*|| = 20 MBytes

$||S||$ 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.

5.4. 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 6 gives the resulting response times for $||S||$ = 2 MBytes, while Figure 7 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 dominates the others. The most significant difference between Figure 6 and the response times obtained in the previous experiment (Figure 4) 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 7 with that obtained in the previous experiment (Figure 5). 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 7 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

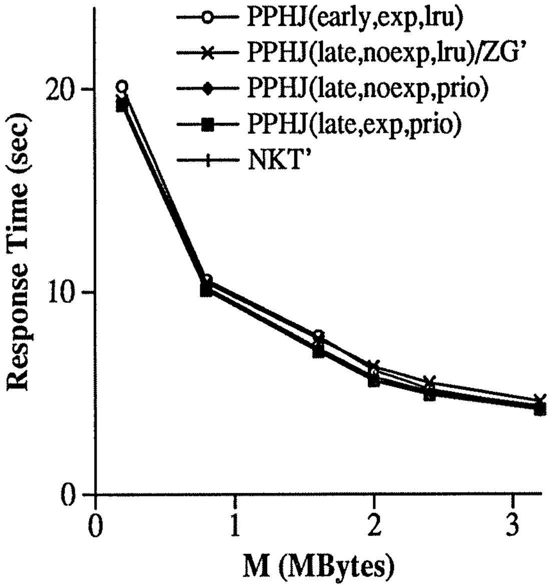

Figure 6: ||*R*|| = ||*S*|| = 2 MBytes

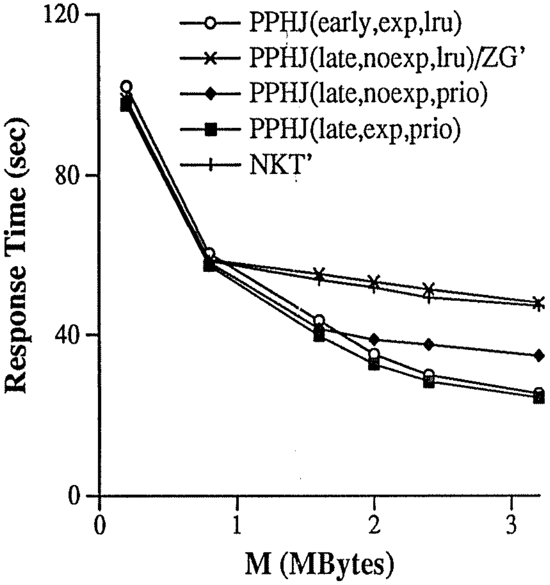

Figure 7: ||*R*|| = 2 MBytes, ||*S*|| = 20 MBytes

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 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 6 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.

5.5. 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 Table 6.

For the first experiment, $||S||$ is set to 2 MBytes, the same size as $||R||$. Figure 8 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 8 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 9. 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.

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

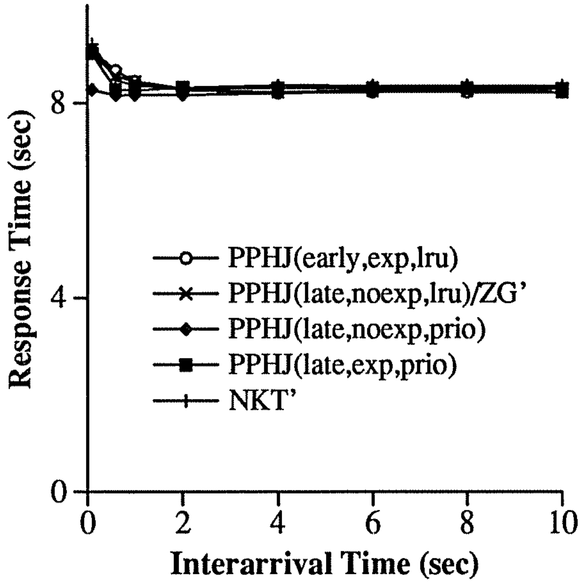

Figure 8: $\|R\| = \|S\| = 2$ MBytes

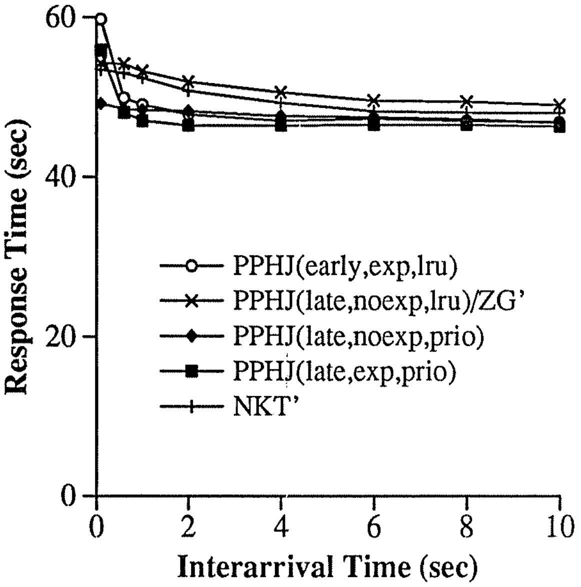

Figure 9: $\|R\| = 2$ MBytes, $\|S\| = 20$ MBytes

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 par-

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

titions 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

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

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

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.

5.6. Discussion of Other Alternatives

As described in Section 3, 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 this 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 the introduction, 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.

6. Conclusion

In this paper, 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 or goal-oriented 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 are insufficient to hold all 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 constructed a detailed DBMS simulation model. Through a series of 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. These findings are important in two ways. First, previous studies [Naka88, Zell90] have proposed algorithms that rely 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. In short, we have identified a simple and yet effective way to deal with memory fluctuations — namely, PPHJ with expansion.

References

[Abbo88] 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.

[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. 16,4*, 1977.

[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 1984 SIGMOD Conf.*, June 1984.

[DeWi90] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.-I Hsiao, R. Rasmussen, "The Gamma Database Machine Project", *IEEE Trans. on Knowledge and Data Engineering, Vol. 2,1*, March 1990.

[Ferg93] D. Ferguson, C. Nikolaou, L. Georgiadis, "Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems", *Proc. of the 2nd Int. Conf. on Parallel and Distributed Information Systems*, January 1989.

[Hari90] J. Haritsa, M. Carey, M. Livny, "On Being Optimistic about Real-Time Constraints", *Proc. of the 1990 ACM PODS Symposium*, April 1990.

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

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

[Kits83] M. Kitsuregawa, H. Tanaka, T. Moto-oka, "Application of Hash to Data Base Machine and Its Architecture", *New Generation Computing, Vol. 1,1*, 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.

[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.

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

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

[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.

[Pang93] H. Pang, M.J. Carey, M. Livny, "Partially Preemptable Hash Joins", *Proc. of the ACM 1993 SIGMOD Conf.*, May 1993.

[REAL92] *Real-Time Systems, 4(3)*, Special Issue on Real-Time Databases, September 1992.

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

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

[Shap86] L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems, Vol. 11,3*, September 1986.

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

[Ston81] M. Stonebraker, "Operating System Support for Database Management", *Comm. of the ACM, Vol. 24,7*, 1981.

[Teng84] J. Teng, R.A. Gumaer, "Managing IBM Database 2 Buffers to Maximize Performance", *IBM Systems Journal, Vol. 23,2*, 1984.

[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.