# Memory-Contention Responsive Hash Joins

Diane L. Davison

University of Colorado at Boulder
Computer Science Department
Campus Box 430
Boulder, Colorado 80309-0430

Goetz Graefe

Portland State University
Computer Science Department
P.O. Box 751
Portland, Oregon 97207-0751

CU-CS-682-93 December 1993

# Memory-Contention Responsive Hash Joins

Diane L. Davison
davison @ cs.colorado.edu
University of Colorado at Boulder, Computer Science Department, Campus Box 430, Boulder, Colorado 80309-0430

Goetz Graefe
graefe @ cs.pdx.edu
Portland State University, Computer Science Department, P.O. Box 751, Portland, Oregon 97207-0751

## Abstract

Fluctuations in memory contention during query execution may compromise the effectiveness of previous allocation decisions and result in excessive I/O costs. In order to maximize system performance, memory-intensive algorithms such as hash join must gracefully adapt to variations in available memory. Responsiveness to memory contention is particularly important in systems processing mixed workloads due to the erratic frequency and magnitude of fluctuations. Earlier studies on adaptable hash joins have advocated lowering I/O costs by reducing the volume, or number of pages, of I/O performed. In this paper, we present a group of memory-contention responsive hash joins that lower I/O costs by using a large unit of I/O, or cluster, to reduce the amount of *time* spent on I/O and that *dynamically vary the cluster size* in response to fluctuations in memory availability. These techniques are effective for static as well as variable memory allocations. Our simulation results demonstrate that our techniques provide much better performance and responsiveness than previous algorithms.

## 1. Introduction

Static memory allocation techniques are inadequate for the execution of queries in a multi-user environment due to the system's inability to predict when new requests will arrive. Differences in query complexity and size complicate allocation by creating a mixed workload in which memory demands may vary significantly among requests. The use of parallelism exacerbates the memory allocation problem by drastically increasing the competition for this scarce resource [SSU91]. This unpredictable environment precludes achieving good performance using static memory allocation techniques since an optimal division of memory among competing queries may become sub-optimal very quickly due to fluctuations in memory contention as queries enter and leave the system. To overcome this uncertainty, the system must adapt to fluctuations in memory contention by adjusting previous allocation decisions. This requires that memory-intensive algorithms such as hash join and sort have the capability to gracefully respond to changes in their memory allocation at any time *during* their execution.

In this paper, we show how memory can be most effectively utilized by hash joins whose memory allocation varies during execution, particularly how a large cluster, or unit of I/O, can be exploited for maximum performance and responsiveness in spite of memory fluctuations. We designed and prototyped a group of memory-contention responsive hash joins and found that fluctuating memory allocations can best be handled by dynamically adjusting the cluster size and

enlarging or reducing the size of the output buffers for spilled partitions based on memory availability.

Most previous hash joins have tried to reduce I/O cost by minimizing the overall I/O volume, or number of pages of I/O. These join algorithms attempt to reduce the time spent on I/O by keeping resident as much of the build input as possible in order to reduce the total amount of I/O performed. However, previous research for joins with *static* memory allocations has shown that it is more effective to use a large unit of I/O, or cluster, to reduce *I/O time* even if it increases I/O volume [Bra84]. Larger clusters reduce the number of I/O calls and thus reduce seek time and rotational latency, the most expensive components of an I/O operation.

It would be possible to simply use a large, fixed cluster size that is independent of the current memory allocation to try to realize the benefit of large I/Os, but this approach has two problems compared to the dynamic approach. First, this could have an adverse effect on the partitioning fan-out. A large cluster size increases the memory commitment for each output buffer and could, therefore, restrict the fan-out and result in large partitions that require multiple levels of partitioning. Second, the most effective cluster size depends on the join's memory allocation as well as the performance of the I/O subsystem. A fixed cluster size does not allow the join to adapt to changes in its memory allocation because it doesn't take either of these factors into account.

In Section 2, we review earlier research on hash join algorithms. We present our memory-contention responsive hash join algorithm, discuss how it responds to memory fluctuations, and describe several variations of the basic algorithm in Section 3. Other adaptable hash join algorithms included in our performance study are discussed in Section 4. We describe our simulator and simulation parameters in Section 5 before offering our experimental results in Section 6. In the final section, we summarize the paper and present our conclusions.

## 2. Related Work

In this section, we review previous research that is related to our work. Before beginning our discussion, we introduce the notation and terminology that we will use throughout this paper.

The join's memory allocation, $M$, is allocated in fixed-size pages. The unit of disk I/O is a cluster, $C$, that is composed of one or more pages. The number of partitions into which the inputs are divided is referred to as the fan-out, $F$. Note that $F$ refers to the total number of partitions, any of which may be resident in memory or spilled to disk. The build input is designated $R$, the probe input $S$, and the number of pages in the input buffer $I$. To account for hash table overhead, we use a fudge factor *fudge*, thus $fudge \times R$ pages of memory are required to entirely contain the hash table for the join.

A hash join is a sequence of one or more steps, where each step processes a pair of build and probe inputs. A step can be the initial partitioning of the base inputs, an intermediate partitioning level (if multiple levels are required), or the in-memory join in the deepest recursion level. A step is composed of the *build stage*, during which the build input is processed, and the *probe stage*, during which the corresponding probe input is processed. Depending on memory availability, a partition may be in one of three states. A partition is *resident* if all build tuples

currently assigned to the partition are entirely contained in the partition's in-memory buffer. If some or all of the partition's build tuples have been written to disk, the partition is *spilled*. If the join's memory allocation increases, the build file of a spilled partition could be read back into memory or *restored* during the probe stage, so that it exists both in memory and on disk. Since at least one page is required per partition and $I$ pages are required for the input buffer, the minimum memory requirement for a hash join is $F + I$ pages, and the join cannot reduce its memory consumption below this amount. In this minimum allocation situation, it is likely that all $F$ partitions would be spilled.

Early work on hash joins assumed a fixed memory allocation for the duration of the join [Bra84, DKO84, KTM83, KNT89, NKT88, Sha86]. Hybrid hash join can be used when the memory allocation is large enough that $R$ can be divided into partitions no larger than memory, i.e., $fudge \times R \leq F \times M$ [DKO84, Sha86]. If there is sufficient memory, hybrid hash join will keep one partition resident to reduce the I/O volume. Nakayama et al. devised a new technique called bucket tuning in which tuples are partitioned into a large number of buckets [NKT88]. Any resident buckets are then grouped into a single resident partition, and spilled buckets are grouped into memory-size partitions for subsequent steps.

Zeller and Gray were the first to propose a hash join that can adapt to variations in available memory during the build stage [ZeG90]. Their adaptable hash join creates a large number of buckets, a number of partitions, and pre-assigns one or more buckets to each of the partitions. They also proposed using a large cluster size for partition buffers, but $C$ (as well as $F$ and the number of buckets) is provided by the optimizer and the paper did not elaborate on how it might be determined. As opposed to hybrid hash join, all partitions are initially resident and each partition is assigned a single cluster for a hash table. During the build stage, each tuple is assigned to a bucket and implicitly to that bucket's partition. Resident partitions may grow in size until the join's memory is exhausted; at that time, the partition requesting additional memory is spilled and retains only one cluster as an output buffer. If a partition ever grows in size beyond the join's current memory allocation, the fan-out is dynamically increased by splitting the large partition into two smaller partitions. Unfortunately, this means that pages associated with the original larger partition must be read at least twice in later steps. The algorithm performs only one level of partitioning and resorts to a hashed-loops algorithm for partitions that are larger than memory. Additional memory can be used during the build stage to enlarge resident partitions but is not exploited during the probe stage. In response to a decrease in memory during either stage, one or more partitions will be spilled. If this is an insufficient reduction in memory usage, the cluster size is decreased; however, the cluster size is never increased in response to an increase in memory.

More recently, Pang et al. proposed a partially preemptible hash join (PPHJ) that adapts to memory fluctuations during both the build and probe stages [PCL93]. PPHJ uses a small, fixed cluster size of $C = 1$ page and a fan-out of $F = \sqrt{fudge \times R}$ that results in partitions of average

size $\sqrt{fudge \times R}$. PPHJ requires a minimum of $\sqrt{fudge \times R}$ pages[1] of memory; therefore, only one level of partitioning is required since the partition files are no larger than memory. Three techniques were presented to allow a join to adapt to memory fluctuations: preassigning clusters to partitions versus assigning clusters to partitions on demand (early vs. late contraction), assigning excess memory to an I/O buffer managed by LRU versus priority policy, and restoring spilled build partitions (expansion). Using late contraction, the same technique used in [ZeG90], all partitions are initially resident and have one cluster assigned for a hash table. As long as sufficient memory is available, partitions may obtain new clusters and grow in size. When a partition needs additional memory but none is available, the resident partition with the highest partition number is spilled and retains only one cluster as an output buffer. The choice of spilling from highest to lowest partition number (i.e., partition $F$ is spilled first and partition 1 is spilled last) is in contrast to dynamically choosing the resident partition to spill based on the current partition size as is done in [NKT88]. A decrease in memory during either the build or the probe stage causes the algorithm to spill one or more build partitions. Additional memory during the build stage is used to expand either resident partitions or the I/O buffer, while additional memory during the probe stage is used either to expand the I/O buffer or to restore spilled build partitions, if that option is enabled. Pang et al. performed a simulation study comparing PPHJ to previous techniques and found that restoration of spilled build partitions during the probe stage is very effective when the inputs differ in size, whereas late contraction and priority spooling provide only a small improvement. The only situation in which restoration was found to harm performance is when the join's memory allocation fluctuates very rapidly.

An unfortunate limitation of PPHJ is that it fails the "Guy Lohman test for join techniques" [Gra93b], requiring that a join algorithm apply to joining three inputs without interrupting the dataflow between the join operators. PPHJ's choice of fan-out and inability to recursively partition the inputs require that it know the exact size of the build input at the beginning of the algorithm. Consider input size under-estimation. In this case, PPHJ would create partitions that are larger than its minimum memory requirement. The algorithm cannot handle this situation, given its minimum memory allocation. Since intermediate result size estimation could be inexact by as much as one or two orders of magnitude [IoC91], this requirement for foreknowledge of the exact input size is a serious limitation. Furthermore, even if the exact input size were known, data skew could also result in partitions larger than the minimum memory requirement.

There are two approaches to reducing I/O cost: reduce I/O volume, or reduce the number of I/O operations. All of the algorithms we have discussed, *static or adaptable*, take the former approach. They attempt to achieve cost savings through reduction of I/O volume by keeping data resident when possible, and maximize memory utilization by creating partitions equal in size to the join's memory allocation. However, the real issue is reducing I/O time, and this involves a tradeoff between I/O volume and the number of I/O operations. It has been shown for both sort

---

[1] PPHJ uses an input buffer of one page that is considered to be overhead rather than part of the join's memory allocation.

and hash-based algorithms using modern disks and having static memory allocations that it is much more effective to use large clusters, or I/O buffers, to reduce *I/O time* even if it increases I/O volume [Bra84, Gra93b, GLS94].

In [Gra90], we explained how these two approaches to I/O cost minimization conflict for external sorting. We described in detail how to determine the optimal cluster size $C$ to be used in conjunction with the maximal fan-in $F = \lfloor M / C - 1 \rfloor$ so that these conflicting goals are balanced[2]. Since merging in sort-merge and partitioning in hash join are dual operations, the exact same conflict and solution apply to hash joins [Bra84, Gra93b, GLS94, Sal88]. On one hand, a smaller cluster size reduces the amount of memory committed to partition output buffers and allows a larger fan-out, which results in smaller partitions that may remain resident and reduce I/O volume. On the other hand, large clusters reduce the number of I/O calls, but this also restricts the fan-out since more pages are committed to each output buffer. Furthermore, the smaller fan-out causes creation of larger partitions, which can cause an increase in I/O volume because multiple levels of partitioning may be required. Due to space limitations, we refer the reader to [Gra90] for the exact derivation, but we point out here that the optimal cluster size can be approximated independently of the input sizes and depends only on the algorithm's memory allocation and the disk parameters (seek time, rotational latency, and transfer speed). The optimal cluster size derivation in [Gra90] is designed for a *static* memory allocation, and we use it here as a starting point in our dynamic algorithm.

## 3. Responsive Hash Joins

The effectiveness of an adaptable algorithm can be determined by two metrics. First, the algorithm must be capable both of capitalizing on memory increases and of gracefully degrading in the face of memory losses. The algorithm's techniques must be effective for fluctuations that vary drastically in both frequency and magnitude. Second, the algorithm must exhibit good responsiveness to reduction requests from the memory manager. We define *responsiveness* as the ability of an algorithm to reduce quickly its memory usage when requested to do so by the memory manager. Responsiveness may assist the memory manager in its allocation decisions, thus it is important to overall system performance. To accomplish these goals, our hash join dynamically adjusts the cluster size, sets the fan-out independently for each step, and enlarges and reduces partition buffers depending on its memory allocation.

In this section, we present and discuss a new algorithm and describe how it responds to memory fluctuations. Then we describe a number of variations of the basic algorithm and an option that may be used with the dynamic variants.

---

[2] Using the maximal fan-in or fan-out results in statically assigning one cluster to the input buffer and one cluster to each partition.

## Basic Algorithm Description

Pseudo-code for the basic algorithm is given in Figure 1. The hash join is composed of a sequence of one or more steps, and the pseudo-code illustrates one step. Recall that a step processes one pair of build and probe inputs and that there will be multiple steps if the build input is larger than memory. The same logic is used to process both base inputs and partition files, so the

```
/* step initialization */
determine fan-out and initial cluster size
obtain one cluster for the input buffer
for each partition
   set state to resident
   obtain one page for a hash table
initialize the free list with all unused pages

/* build stage */
for each build tuple
   hash the tuple to a partition
   while the tuple has not been added to the partition
      if the tuple fits in the partition
         insert the tuple into the partition
      else if a new page is available
         enlarge the partition with a new page[3]
      else if the partition is resident
         spill the largest resident partition
         return all pages except one to the free list
      else   /* partition is spilled */
         flush the partition output buffer

/* probe stage */
for each probe tuple
   hash the tuple to a partition
   if the partition is resident
      probe the hash table
      if there is a match, emit the result
   else   /* the partition is spilled */
      if the tuple does not fit in the partition
         if a new page is available
            enlarge the partition with a new page[3]
         else
            flush the partition output buffer
      insert the tuple into the partition's output buffer
```

Figure 1. Memory-Contention Responsive Hash Join — Basic Algorithm.

---

[3] If the partition is spilled, the size of its output buffer is limited to $C$. See the discussion in the text.

inputs will be recursively partitioned if necessary. Thus, our algorithm does not depend on accurate estimates of the input sizes. While Zeller and Gray's algorithm also handles inaccurate estimates of input sizes, their algorithm resorts to a hashed-loops algorithm if the initial level of partitioning produces partitions larger than memory.

At the beginning of a step, it is possible to set the cluster size and fan-out as desired since no data are resident in memory. Here we could ideally balance the I/O volume and number of I/O operations by using the optimal cluster size and the maximal fan-out based on the current memory allocation, but other considerations may affect these choices. Since extremely large I/O requests would monopolize the disk, we limit all I/O requests to $IO_{max}$. A reasonable choice for $IO_{max}$ is one track, and that is the value we use. The cluster size $C$ is set to the smaller of $IO_{max}$ and the optimal cluster size that was discussed in Section 2. The maximal fan-out of $F = \lfloor M / C - 1 \rfloor$ could be very large, resulting in a join that is less resilient to memory loss due to a larger minimum memory requirement. Therefore, rather than using the maximal fan-out, we base the fan-out on the size of the build input. For the first step, the fan-out is set to $F = \sqrt{fudge \times R_{est}}$, based on the estimated size of the base build input in pages. Notice that $R_{est}$ is an estimate of the size of $R$; the algorithm benefits from but does not depend on estimation accuracy. For all other steps, F is set to $\sqrt{fudge \times R_i}$, where $R_i$ pages is the actual size of the build partition file for the step. F will be smaller for later steps since the partition files are smaller. For example, consider a join for which $R_{est} = 10000$ pages and $fudge = 1.2$. For the first step, $F = \sqrt{1.2 \times 10000} = 110$. If $R_{est}$ were low by 10% (i.e., $R = 11000$ pages), the partition files $R_i$ would be approximately of size 100 pages. Then for subsequent steps, the fan-out would be $F = \sqrt{1.2 \times 100} = 11$. The fan-out for our first step, $F = \sqrt{fudge \times R_{est}}$, is the same as the fan-out that PPHJ uses for its first step, except that PPHJ requires $R_{est}$ to be accurate. PPHJ uses classic in-memory hash join in subsequent steps, so it sets the fan-out for the first step only. All of the $F$ partitions are resident initially and are given a one-page buffer, but additional pages may be obtained later as needed (this technique was used in [ZeG90] and also [PCL93], where it was referred to as late contraction). Any unused pages are added to the operator's free list.

During the build stage, all build tuples are consumed, and each is assigned to a partition, $P_i$. A tuple that is assigned to a resident partition is copied to the partition's buffer and inserted into the hash table. If there is no space for the tuple in the partition's buffer, a new page may be obtained to enlarge this resident partition's buffer. If no new page is available, the largest resident partition is spilled and retains only a single page as an output buffer. The join spills the largest resident partition since it makes available the most memory, similar to [NKT88]. Moreover, spilling the largest resident partition is the most reasonable action if nothing is known about hash value skew in the probe input [Gra93a, NKT88]. After the largest partition has been spilled, either $P_i$ is still resident and may obtain an additional page, or $P_i$ is spilled and has space in its output buffer. A tuple assigned to a spilled partition is copied to the partition's output buffer if there is space. If the buffer is full, the join will *enlarge the spilled partition buffer* by obtaining a new page if one is available, or by flushing the output buffer if none is available. While resident partitions may be enlarged without limit, spilled partition buffers may not exceed the cluster size. Figure 2 illustrates both minimal and enlarged spill buffers. The dashed box represents the cluster size, which is the target size for buffers of spilled partitions. In Figure 2, partition 1 has a

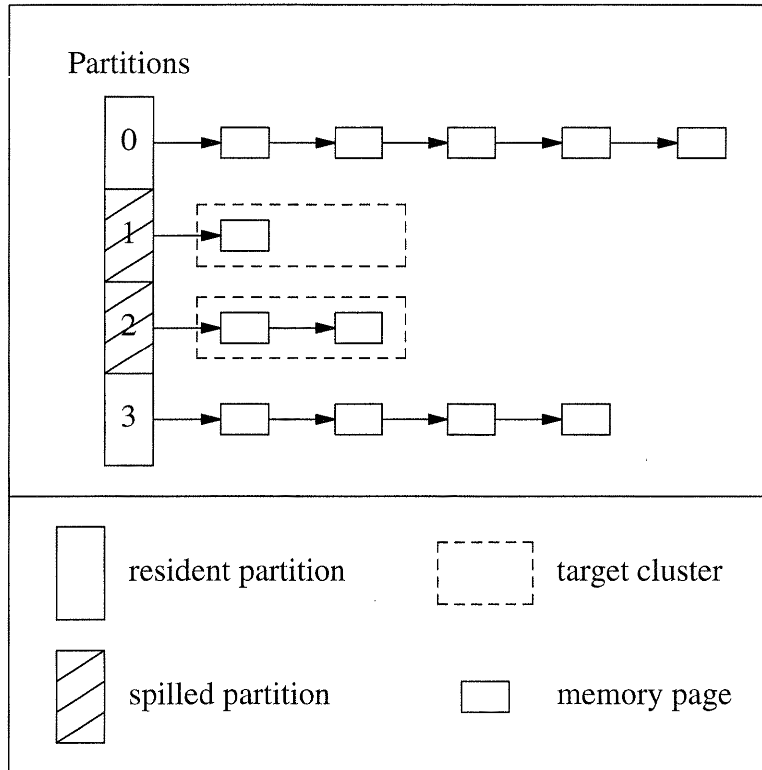


Figure 2. Partitioning with Dynamic Clusters.

minimal spill buffer of 1 page, while the spill buffer for partition 2 has been enlarged to the current cluster size of 2 pages. At the end of the build stage, all spilled partitions are flushed and each retains one page as an output buffer to be used while partitioning the probe input.

After the build stage is completed, the probe input is consumed. At the beginning of the probe stage, there are $F$ partitions, some resident and some spilled. For probe tuples assigned to resident partitions, the hash table is probed immediately. A probe tuple that is assigned to a spilled partition is copied to the partition's output buffer, if there is space. If the buffer is full, the partition buffer will be enlarged with a new page if one is available, or flushed if none is available. This enlargment of spilled partition buffers is exactly the same as that during the build stage. At the end of the probe stage, all spilled partitions are flushed and all buffer memory is returned to the operator's free list.

When both build and probe stages for one step are complete, the algorithm initializes the next step to process the next pair of partition files, if any remain to be processed.

To summarize, the basic algorithm manages the tradeoff between reducing I/O volume and reducing the number of I/O requests by having resident partitions and spilled partitions compete for memory. Resident partitions are restricted in size only by memory availability, whereas spilled partition buffers may not exceed one cluster. This differs from previous approaches that focus on reducing the I/O volume and do not enlarge spill buffers in response to increases in

memory.

## Algorithm Discussion: Memory Fluctuations

Before describing how our basic algorithm responds to memory fluctuations, it is important to emphasize the difference in the way we utilize the dynamic cluster size versus the way a fixed cluster size is used. An algorithm with a large, fixed cluster size uses the cluster size as the unit of I/O for all I/O requests. Rather than using the dynamic cluster size as the mandatory unit of I/O, we use it as the *target size* for the output buffers of spilled partitions. That is, the goal is to use output buffers of one cluster in size, but the ability to realize that goal depends on memory availability. The target cluster was illustrated in Figure 2.

A memory fluctuation *between* steps is handled very easily by the basic algorithm, since each step is processed independently and may have a different fan-out $F$ and cluster size $C$. However, a fluctuation *during* a step requires special action.

Additional memory is utilized to enlarge the join's input and output buffers during either the build or probe stage. Although the fan-out remains unchanged for the duration of a step, the cluster size may be adjusted during a step. In response to an increase in memory, the cluster size is recalculated as the minimum of $IO_{max}$, the largest I/O request allowed, and $C = \lfloor M / (F + 1) \rfloor$, the maximal cluster size. The maximal cluster size is the one that would provide an even division of memory among the $F$ partitions and the input buffer. While no buffer is guaranteed $C$ pages of memory, the cluster size is used as the target size for the I/O buffers. Once the new cluster size has been determined, the input buffer is increased to $C$ pages, if possible. A large input buffer is beneficial for all partitions, so it is a wise investment of memory. Any remaining pages after enlargement of the input buffer are assigned to the join's free list so that they may be used to enlarge partition buffers as needed. For example, in Figure 2, additional memory could be obtained by any of partitions 0, 1, or 3; if the additional memory resulted in an increase in the target cluster size, partition 2 could also compete for the memory. In contrast to the use of additional memory in our algorithm, the algorithms of Pang et al. as well as Zeller and Gray use additional memory only to reduce I/O volume by avoiding I/O (Zeller and Gray may use large buffers initially, but additional memory is used only to enlarge resident partitions).

In response to a decrease in memory, the join must reduce its memory usage by reclaiming memory, first from its free list and then from its partition buffers. The join could also reduce the input buffer, but since a large input buffer is so generally beneficial, our join never reduces the size of the input buffer. The disadvantage of retaining a large input buffer is that it increases the join's minimum memory requirement ($F + I$ pages). If the join's reduced memory allocation is already equal to its minimum memory requirement, the join is unable to free any more pages. Otherwise, to decrease its memory usage, the join must reclaim pages until its memory usage has been sufficiently reduced. First, it will reclaim pages from its free list. Second, it will reduce the buffers of spilled partitions to one page by flushing the full pages from the partition output buffers. Third, it will spill resident partitions and reclaim all but one page to be used as an output buffer. Both spilled and resident partitions are chosen in increasing order of partition number. For example, given the situation in Figure 2, a request for the join to decrease its memory usage

to its minimum memory requirement of $4 + I$ pages would cause the following actions: the full pages of partition 2 would be flushed and its buffer reduced to 1 page; partition 0 would be spilled and given one page; and partition 3 would be spilled and given one page. Finally, after memory consumption has been adequately reduced, the new cluster size will be determined in the same manner as for a memory increase. In contrast to our handling of memory decreases, Zeller and Gray spill resident partitions before they consider reducing the cluster size. Pang et al. reclaim excess memory from the join's I/O buffer before they spill resident partitions.

In summary, our algorithm capitalizes on memory increases by enlarging both resident partitions and the output buffers of spilled partitions. It adapts to memory losses by decreasing the sizes of output buffers of spilled partitions as well as spilling resident partitions. By adjusting both resident and spilled partitions in response to memory fluctuations, the algorithm can better balance I/O volume and the number of I/O requests to reduce I/O time. Moreover, a large cluster size should allow an algorithm to be more responsive to memory reduction requests from the memory manager, since algorithms usually require I/O to reduce their memory usage.

## Algorithm Variants

Our basic algorithm provides one approach to utilizing large clusters in a dynamic environment. In this section, we present several variations of the basic algorithm that take different approaches to exploiting large clusters in the face of unpredictable and changing memory allocations.

- basic algorithm (*basic*)

  The basic algorithm, shown in Figure 1, was discussed above. The size of resident partitions is limited only by memory availability, whereas spilled partition buffers may not exceed one cluster.

- immediate allocation of entire cluster (*fair*)

  To provide a more even distribution of memory among the partitions, this variant attempts to give all partitions a *fair share* of memory. Rather than using $C$ only as an upper limit on the size of each output buffer of a spilled partition, this algorithm immediately allocates the buffer to be of size $C$. That is, when a resident partition is spilled, rather than being reduced to one page, it is given $C$ pages. During memory reduction, this algorithm also attempts to evenly decrease the size of partition buffers, first of spilled partitions and then of resident ones.

- maximize write operations (*max*)

  This variant maximizes the size of each write operation by always flushing the spilled partition with the largest current memory allocation and spilling the largest resident partition. When a tuple must be inserted into a spilled partition that has a full buffer and no additional pages are available, all our other algorithm variants flush the partition that currently requires space. Instead, if the partition's buffer is less than $C$ pages, this variant flushes the spilled partition with the largest output buffer and leaves it only one page as an output buffer; the partition with the full buffer may then enlarge its output buffer. In response to a memory decrease, this variant choses the partition with the largest memory allocation to flush or spill, rather than the partition with the lowest partition number.

• statically determine cluster size and fan-out (*stat*)

To provide a basis for comparison with our dynamic techniques, this variant is an adaptation of what has been shown to be effective for joins with static memory allocations. It *statically* sets the optimal cluster size and fan-out based on the join's memory allocation at algorithm initialization time and retains these initial settings throughout its execution. Rather than preassigning one cluster to each partition, *stat* allows resident partitions to grow without limit and gives one cluster to each spilled partition. In response to memory fluctuations, *stat* behaves exactly like *basic*, except it does not adjust the cluster size.

For our dynamic algorithm variants (*basic*, *fair*, *max*), we provide the following option:

• balance cluster size with restoration (*bal*)

It is clear that large clusters are effective for static memory allocations and that restoration can be effective in some situations. To derive benefit from both techniques, large dynamic buffers and restoration, we provide an option that attempts to balance the two. An algorithm with the *bal* option responds to an increase in memory during the probe stage as follows. It first attempts to realize the largest part of the cost savings from a large cluster size, and then it uses any remaining memory to restore as many spilled build partitions as possible. Figure 3 shows the general shape of the join cost curve as the cluster size increases. To realize the largest part of the cost savings from a large cluster size, the algorithm simply chooses a cluster size where the cost curve begins to flatten. Such a point is illustrated in Figure 3. The algorithm preserves enough memory so that each partition could be of this cluster size, and uses any remaining memory for restoration.

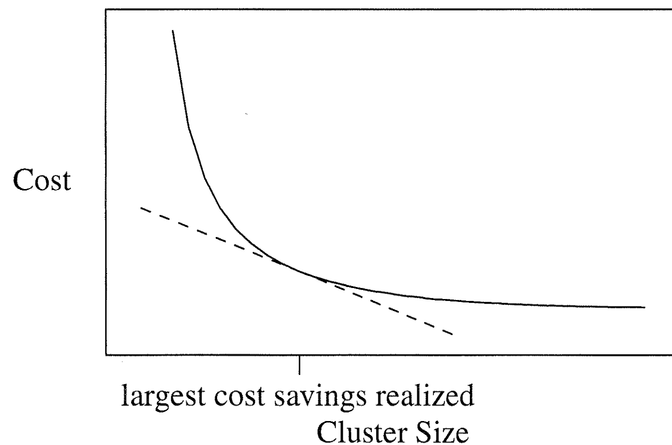


Figure 3. Cluster Size with Most of the Performance Gain.

# 4. Other Adaptable Algorithms

A thorough simulation study demonstrated that PPHJ has as good or much better performance than previous memory-adaptable algorithms, including adaptable variants of Grace and hybrid hash join, and Zeller and Gray's adaptable hash join algorithm [PCL93]. Therefore, it is sufficient to compare our algorithms only to PPHJ. We discussed PPHJ in detail in Section 2, and in this section we describe our implementation of PPHJ. We implemented PPHJ as faithfully as possible based on the algorithm description in [PCL93], with one important modification to allow PPHJ to handle pipelined input.

The original PPHJ algorithm requires foreknowledge of the exact build input size to set the fan-out, and requires that tuples are evenly distributed. We modified the PPHJ algorithm to handle pipelined input to make it comparable to our algorithms. We refer to the modified algorithm as $\mathrm{PPHJ_p}$. For steps after the initial step, PPHJ uses a different cost function that does not use copying. Since we do not assume an accurate estimate of the build input size and do not require tuples to be evenly distributed, we did not use this simplification. $\mathrm{PPHJ_p}$ recursively partitions its inputs and processes partition files exactly the same way it processes the base inputs. That is, rather than reading a build partition file, $R_i$, and inserting the tuples into the hash table without copying, $\mathrm{PPHJ_p}$ hashes tuples and copies them to the appropriate partition. For steps processing partition files, $\mathrm{PPHJ_p}$ uses the same fan-out as our dynamic algorithms, $F = \sqrt{fudge \times R_i}$.

Among the variants of PPHJ studied in [PCL93], the most effective variant combines late contraction, restoration (expansion), and priority spooling, with restoration providing the largest performance improvement. $\mathrm{PPHJ_p}$ includes both late contraction and "enhanced" LRU spooling, and it includes restoration as an option. Since prioritized spooling provides only a small improvement, we use LRU spooling with the enhancement that we prefer build pages over probe pages. PPHJ provides this enhancement as part of prioritized spooling, so our spooling technique should be slightly better than LRU spooling, but not quite as good as prioritized spooling. Restoration was effective in some but not all situations analyzed in [PCL93], so we include $\mathrm{PPHJ_p}$ both with and without restoration (*res* option).

# 5. Database Simulator

We implemented a simulator to study how to most effectively adapt to memory fluctuations in a dynamic environment, and to determine the effectiveness of our techniques compared to previous approaches. In this section, we describe the simulated hardware architecture and software architecture.

The simulated machine has a single CPU, a single disk, a memory manager, and a query source. Table 1 summarizes the machine architecture, with the disk paramaters taken from a Maxtor MXT-1240S. To prevent any request from monopolizing the disk, all I/O requests are limited to $\mathrm{IO}_{max}$, one track. If the current I/O request is for the next cluster in the same file as the previous request, the I/O is charged as sequential (*latency* + *transfer*); otherwise, it is charged as random (*seek* + *latency* + *transfer*). We use synchronous I/O in this simulation because it provides a better measure of total resource consumption.

| Architecture Parameter | Value |
|---|---|
| CPU Speed | 20 MIPS |
| Page Size | 8 KB |
| Avg. Disk Transfer Rate | 3.5 MB/sec |
| Avg. Disk Rotational Latency | 4.76 ms |
| Avg. Disk Seek Time | 8.5 ms |
| Avg. Disk Track Size | 256 KB |
| $IO_{max}$ | 256 KB |

Table 1. Machine Architecture.

The join we model is a primary key-foreign key join, and each probe tuple matches with exactly one build tuple. The hash values are assumed to follow a uniform distribution. Table 2 lists the number of CPU instructions for the simulator operations[4]. The fudge factor to account for hash table overhead is $fudge = 1.2$.

The simulation consists of a single stream of adaptable two-way join queries. At any point in time, one adaptable join is active; as soon as that join completes, another one is started. Concurrent operations are reflected in the fluctuating memory allocations. The join has a minimum memory requirement of $F + I$ pages, providing an input buffer and one page per partition as an output buffer. A join that is allocated less memory than its minimum memory requirement is suspended until additional memory is available. The memory allocated to a join is considered to be "pinned" in the buffer and managed entirely by the join operator without system intervention. The memory manager may decrease or increase the join's memory allocation, and the join

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

Table 2. Number of CPU Instructions per Operation.

[4] This table was reproduced from [PCL93].

responds to these changes by unpinning or pinning pages. For this simulation, the join responds as soon as possible to memory fluctuations. For example, if the join is in the process of reducing its memory consumption and more memory becomes available, it will cease reduction.

## 6. Experimental Results

In this section, we evaluate our techniques compared to $PPHJ_p$ for several different workloads. To facilitate comparison with $PPHJ_p$, we have structured our simulation experiments similarly to those presented in [PCL93]. However, in our experiments, we consider that the adaptable join may be part of a multi-join query with pipelined inputs, so we measure only the processor and I/O time used by the join. The performance metric we use is average join response time, which includes the CPU time used by the join and the time used for I/O to partition files. Note that this is not the same as the average query response time since we specifically exclude the time to read the base inputs and write the final output. In this section, we refer to the metric as average response time, but the reader should keep in mind that the measurement is the join's contribution to the response time rather than the overall query response time. We report the I/O activity as the average number of I/O requests and the average I/O volume in pages. The I/O volume is further differentiated as I/O to either build or probe partition files. The adaptable join algorithms and options are summarized in Table 3. The first experiment demonstrates the ability of the join algorithms to adapt to a large change in memory availability in an otherwise stable system. This experiment shows how well the algorithms can overcome planning based on a poor prediction in the amount of memory available to the join, and also illustrates the responsiveness of the algorithms. We examine how well the algorithms adapt under different magnitudes of memory contention in the second experiment in Section 6.2. The final experiment in Section

| Algorithm Variant | Description |
|---|---|
| basic | Limit buffers to C<br>adjust C at every fluctuation |
| fair | Force buffers to be C pages<br>adjust C at every fluctuation |
| max | Maximize write requests<br>adjust C at every fluctuation |
| stat | Statically set C & F |
| $PPHJ_p$ | Partially Preemptible Hash Join |
| **Options** | **Description** |
| bal | Balance cluster size and<br>restoration (basic, fair, max only) |
| res | Restore ($PPHJ_p$ only) |

Table 3. Algorithms and Options.

6. 3 shows the stability of the algorithms with respect to different frequencies of fluctuations.

## 6.1. Poorly Predicted But Stable Memory Allocation

This experiment demonstrates the ability of a join algorithm to adapt to a large change in its memory allocation. Such a fluctuation might occur if a large query completes or begins during the execution of the join, but system activity is otherwise stable until the join completes. The join is given an initial allocation of memory that it uses for planning purposes, i.e., to determine the cluster size and fan-out. This allocation has a very short duration with a mean of 1 second after which the join experiences either a large increase or a large decrease in its memory allocation. After the single fluctutation, the join's memory allocation remains stable and the join completes with that final memory allocation. The join inputs for this experiment are $R = 5MB$ and $S = 50MB$. Obviously, a shorter or longer duration than 1 second would produce different results. We simply chose a duration that would allow us to examine fundamental differences among the algorithms. The next section illustrates the more general situation of many fluctuations with varying durations.

Figure 4 shows the average response time for each of the algorithms when the join experiences a large memory increase. In this experiment, the join's memory is increased from 1 MB to 4 MB. Notice that this memory is the join's memory allocation rather than the total machine memory, thus this is not a memory-starved environment.

All of our variants provide much better performance in response to the large increase than $PPHJ_p$. The reason for this is clear from Table 4. Even though $PPHJ_p$ has a total I/O volume that is much smaller than our variants, its number of I/O calls is much larger due to its small cluster size. $PPHJ_p$:res reduces the I/O volume from $PPHJ_p$, but even $PPHJ_p$:res has a response time that is 50% higher than our worst dynamic variant *fair* (94 vs. 62 seconds).

From Table 4, we see that the *bal* option, which combines large clusters with restoration, is clearly beneficial for *basic* and *max*. It allows the algorithms to realize the enormous savings in

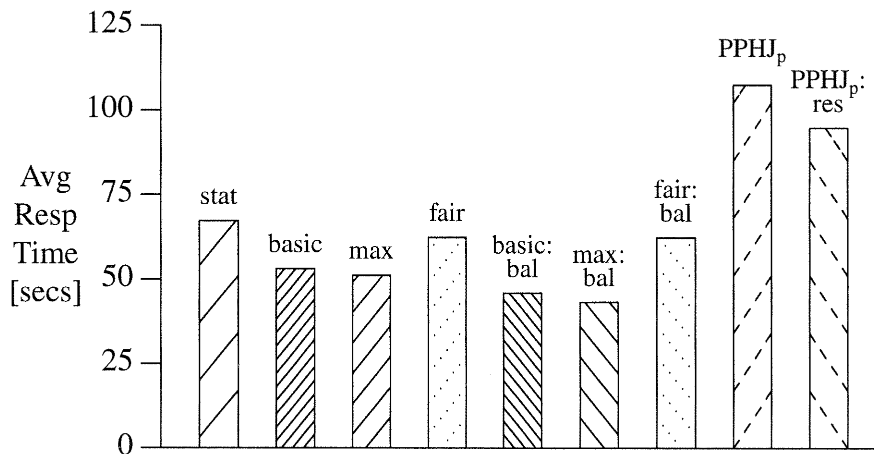


Figure 4. Large Memory Increase (1*MB to* 4*MB*).

| Alg. Variant | Average I/O | | | | Average Response Time [seconds] |
|---|---|---|---|---|---|
| | Total $IO_{calls}$ | Total $IO_{vol}$ [pages] | Build $IO_{vol}$ [pages] | Probe $IO_{vol}$ [pages] | |
| stat | 1792 | 14214 | 1306 | 12908 | 67.30 |
| basic | 691 | 12456 | 1152 | 11304 | 53.14 |
| max | 642 | 12010 | 1110 | 10900 | 51.21 |
| fair | 762 | 15496 | 1430 | 14066 | 62.34 |
| basic:bal | 703 | 9758 | 1168 | 8590 | 45.85 |
| max:bal | 552 | 9536 | 1122 | 8414 | 43.13 |
| fair:bal | 758 | 15464 | 1448 | 14016 | 62.21 |
| $PPHJ_p$ | 7922 | 7922 | 742 | 7180 | 107.38 |
| $PPHJ_p$:res | 6910 | 6910 | 742 | 6168 | 94.69 |

Table 4. IOs for Large Memory Increase.

the number of I/O calls compared to $PPHJ_p$, and also to reduce the total I/O volume by avoiding some probe I/O compared to our variants without this option.

Overall, *stat* has the highest response time of our variants due to its inability to adapt to the memory gain by increasing its cluster size. This results in a significantly higher number of I/O calls than our other variants. *Fair* is clearly the worst performer of our dynamic variants. In evenly dividing memory, *fair* attempts to make an equitable division that penalizes no partition. Unfortunately, this even division results in poor memory utilization. After a spilled partition is flushed, most of its buffer pages are idle for a significant amount of time. These pages could be used immediately by resident partitions to reduce the I/O volume, but this is not possible with *fair* since it ensures each partition a fair share of memory. This ineffective utilization of memory also prevents the *bal* option from providing any benefit in this situation, since the memory is not available for restoration. Our other variants (*basic* and *max*) have similar response times. However, the technique used by *max* to perform larger writes results in a somewhat lower response time than *basic*. This is the case for the two algorithms both with and without the *bal* option.

For the second part of this experiment, we examine the ability of the join algorithms to adapt to a large reduction in memory. In addition to the effectiveness at reducing response time, we also examine the algorithm responsiveness. In this experiment, the join's memory allocation is decreased from 4 MB to 1 MB.

Figure 5 shows the average response time for each of the algorithms when the join experiences a large memory decrease. The average I/O activity is shown in Table 5. Algorithm variants with either the *res* or *bal* option are not included in this experiment since restoration is not possible in response to a memory decrease.

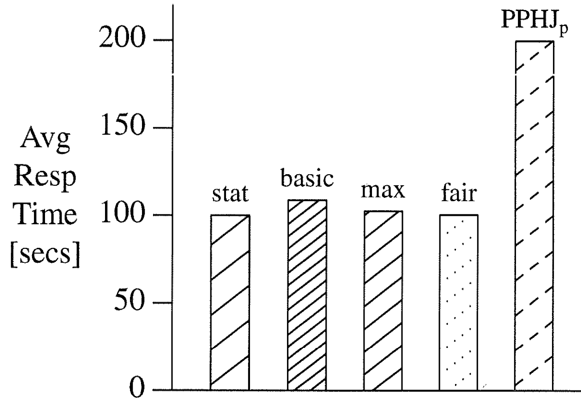


Figure 5. Large Memory Decrease (4*MB to* 1*MB*).

| Alg. Variant | Average I/O | | | | Average Resp. Time [seconds] |
|---|---|---|---|---|---|
| | Total $IO_{calls}$ | Total $IO_{vol}$ [pages] | Build $IO_{vol}$ [pages] | Probe $IO_{vol}$ [pages] | |
| stat | 3302 | 17392 | 1606 | 15786 | 100.00 |
| basic | 3999 | 16738 | 1546 | 15192 | 108.76 |
| max | 3582 | 16544 | 1530 | 15014 | 102.46 |
| fair | 3394 | 16786 | 1548 | 15238 | 100.50 |
| $PPHJ_p$ | 15554 | 15554 | 1436 | 14118 | 199.55 |

Table 5. IOs for Large Memory Decrease.

$PPHJ_p$ has a slightly lower I/O volume than our variants, but its large number of I/O calls contributes heavily to its higher response time. Our variants make one fourth to one fifth the number of I/O calls made by $PPHJ_p$, resulting in response times that are 80-100% faster than $PPHJ_p$.

In this situation, all of our variants provide similar response times. Although *stat* and *fair* provided the worst performance in reponse to the memory increase, here they provide similar response times to *max*. While they have a slightly higher I/O volume than *max*, they have a slightly lower number of I/O calls. The preallocation of one cluster to each spilled partition leaves less memory for resident partitions, increasing the I/O volume. Having fewer resident partitions results in more memory for output buffers of spilled partitions, decreasing the number of I/O calls. Thus, the technique of preallocation is effective in response to memory decreases, although it was ineffective for memory increases. *Max*, which was our most effective variant for memory increases, provides similar performance to *fair* and *stat*. Furthermore, *max* is successful at reducing slightly the number of I/O calls and thus the response time compared to *basic*.

In Section 3, we defined responsiveness as the ability of an algorithm to reduce quickly its

| Alg. Variant | Avg. Response Delay Total [seconds] | Per MB [seconds] |
|---|---|---|
| stat | 0.64 | 0.46 |
| basic | 0.90 | 0.40 |
| max | 0.91 | 0.40 |
| fair | 0.66 | 0.44 |
| $PPHJ_p$ | 2.36 | 1.03 |

Table 6. Algorithm Responsiveness.

memory usage. In Table 6, we show the responsiveness of the algorithms for the last experiment, the large memory decrease that was shown in Figure 5 and Table 5. We measured responsiveness as the average response delay for the algorithm to reduce its memory usage from 4 MB to 1 MB. Table 6 shows the total delay for each algorithm to complete the memory reduction, as well as the delay specified as seconds per megabyte. The total delay shows that the amount of time required to reduce memory usage by 3 MB varies considerably among the different algorithm variants. The explanation for this is that at any point in time, different numbers of partitions may be spilled depending on the techniques employed by the particular variant. Thus, the amount of work (and time) necessary to reduce memory usage may vary. For this reason, it is more logical to examine the delay per megabyte, which takes into account the amount of data as well as the amount of time. From the delay per megabyte column in Table 6, we see that all of our variants provide a much quicker response than $PPHJ_p$. Our most responsive variants, *max* and *basic*, are over 150% faster than $PPHJ_p$. Thus, our techniques of utilizing large, dynamic units of I/O provide much better algorithm responsiveness.

In summary, this experiment has demonstrated three important points. First, large clusters provide much lower response times than previous techniques for both large increases and large decreases in the memory allocation. This technique can be effectively utilized in combination with the technique of restoration to further reduce response time, as in our *bal* option. Second, the static technique is ineffective in response to memory increases due to its inability to adapt to the gain. Third, our techniques are much more responsive to memory reduction requests than $PPHJ_p$, due to the use of large, dynamic clusters. We also found that overall *max* is our most effective variant in response to both memory increases, memory decreases, and responsiveness. Therefore, the remaining experiments will include our variants *max* and *max:bal* in addition to $PPHJ_p$ and $PPHJ_p$:res.

## 6.2. Varying Contention

In this experiment, we subject the join algorithms to varying magnitudes of contention to determine if the techniques are equally effective at different levels of contention. We attempt to simulate a situation in which the adaptable join competes for memory with other queries in the system. System activity is fairly unpredictable since other queries may enter or leave the system at any time. The join's memory allocation is uniformly selected from the range of 80-100% of

total system memory 80% of the time. This represents changes in available memory as other queries enter and leave the system. The other 20% of the time, the allocation is uniformly selected from the range of 0-100% of total system memory. This represents the possibility of an occasional surge in contention due to either a particularly large query or the simultaneous arrival of several smaller queries. The duration of the join's allocation is based on an exponential distribution with a mean of 1 second. We vary the magnitude of contention by varying the amount of system memory. The join inputs for this experiment are $R = 5MB$ and $S = 50MB$. Figure 6 shows the average response time for the various algorithms where the memory from which the adaptable join's allocation is taken ranges from 0.625 MB to 8 MB. The high contention situation with 0.625 MB may seem to create excessive contention, but it is actually quite reasonable. Hybrid hash join for our inputs requires only 0.22 MB of memory, and our experimental parameters with 0.625 MB of memory permit more than that most of the time. Furthermore, the other end of the range (8 MB) provides extremely low contention. To perform a classic in-memory hash join, our inputs require approximately 6.2 MB of memory, accounting for hash table overhead. Our experimental parameters with 8 MB of memory give the join about 6.4 MB most of the time.

As shown in Figure 6, our algorithm variants produce much lower response times than $PPHJ_p$ at higher levels of contention. At low contention ($M = 8MB$), where the technique of restoration is most effective, *max:bal* is a little faster than $PPHJ_p$:res. We now examine the two extremes of high and low contention in detail, i.e., all measurements in Figure 6 for 0.625 MB

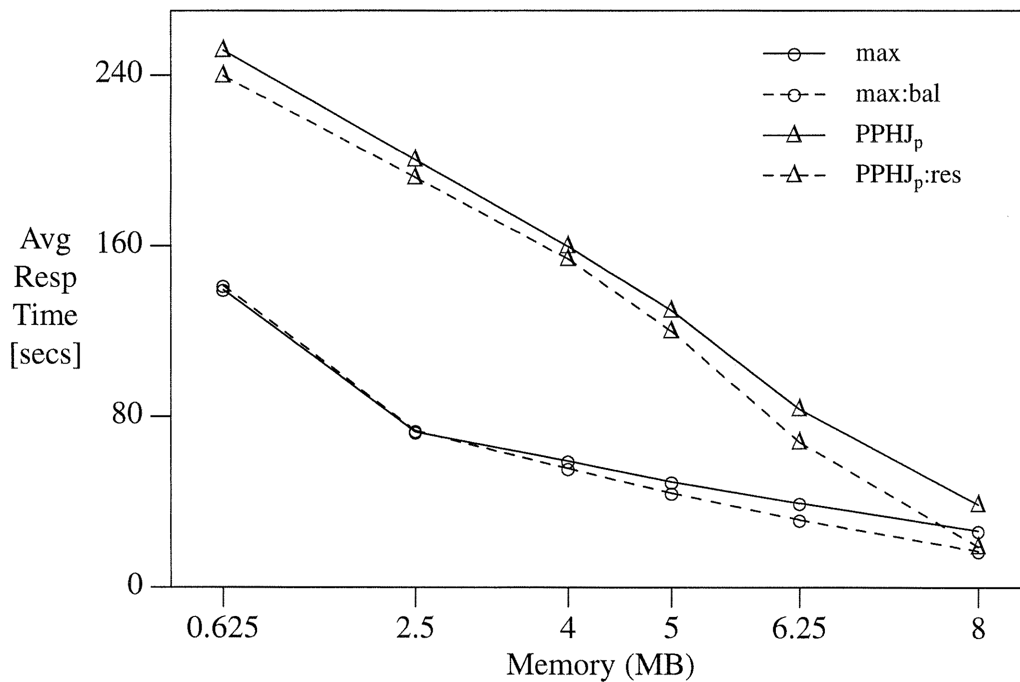


Figure 6. Varying Contention ($R = 5MB, S = 50MB$).

memory and for 8 MB memory.

Detailed data for the high contention situation are shown in Table 7. *Max* and $PPHJ_p$ have a similar I/O volume, but *max* has less than one third as many I/O calls as $PPHJ_p$, resulting in a response time that is 70% faster. The technique of restoration provides only a small improvement under high contention since the scarcity of memory prevents this technique from being used often. For $PPHJ_p$, restoration increases the build I/O volume by 352 pages to decrease the probe I/O volume by 1098 pages, a net win (albeit small). In this situation, *max:bal* provides a similar response time to *max*. The *bal* technique decreases I/O volume by only 99 pages, but increases the number of I/O calls by 296. *Max:bal* uses restoration very little and achieves most of its response time reduction compared to $PPHJ_p$ from large, dynamic clusters. Our techniques are very effective for the high I/O volume resulting from a high contention environment.

For the low contention situation, the I/O activity is shown in Table 8. Our balanced approach is very effective here, causing *max:bal* to be about 55% faster than *max*. Restoration is also very effective, with the result that $PPHJ_p$:res is about 100% faster than $PPHJ_p$. It is interesting to note from Table 8 that the I/O volume for our variants is over twice as high as for the $PPHJ_p$ variants due to our use of large clusters. Using memory for large spill buffers prevents

| Alg. Variant | Average I/O | | | | Average Response Time [seconds] |
|---|---|---|---|---|---|
| | Total $IO_{calls}$ | Total $IO_{vol}$ [pages] | Build $IO_{vol}$ [pages] | Probe $IO_{vol}$ [pages] | |
| max | 5560 | 18032 | 1706 | 16326 | 139.37 |
| max:bal | 5856 | 17933 | 2001 | 15932 | 141.31 |
| $PPHJ_p$ | 18374 | 18374 | 1764 | 16610 | 251.68 |
| $PPHJ_p$:res | 17628 | 17628 | 2116 | 15512 | 239.72 |

Table 7. IOs for High Contention ($M = .625MB$, $R = 5MB$, $S = 50MB$).

| Alg. Variant | Average I/O | | | | Average Response Time [seconds] |
|---|---|---|---|---|---|
| | Total $IO_{calls}$ | Total $IO_{vol}$ [pages] | Build $IO_{vol}$ [pages] | Probe $IO_{vol}$ [pages] | |
| max | 215 | 4958 | 720 | 4238 | 26.51 |
| max:bal | 152 | 1977 | 1227 | 750 | 17.04 |
| $PPHJ_p$ | 2432 | 2432 | 362 | 2070 | 38.93 |
| $PPHJ_p$:res | 953 | 953 | 673 | 280 | 19.10 |

Table 8. IOs for Low Contention ($M = 8MB$, $R = 5MB$, $S = 50MB$).

using the memory for resident partitions, resulting in the increase in I/O volume. However, the larger clusters significantly reduce the number of I/O calls. Comparing *max* and $PPHJ_p$, this tradeoff is very effective (*max* is about 47% faster). For *max:bal* and $PPHJ_p$:res, the tradeoff is also effective (*max:bal* is 12% faster).

We performed an identical experiment to the above with equal join inputs $R = S = 5MB$. Figure 7 shows the results of this experiment. The relative performance of *max* and $PPHJ_p$ is very similar to the previous experiment with different input sizes. The main difference, as was shown in [PCL93], is that the technique of restoration does not reduce the response time (in fact, $PPHJ_p$:res is actually slightly worse that $PPHJ_p$ in some cases). Again, our techniques provide a significant improvement in performance at higher levels of contention, and some improvement at the lowest level of contention. Detailed data for the high contention situation are shown in Table 9. The $PPHJ_p$ variants and our variants have a similar I/O volume, but our variants have about one third as many I/O calls. This reduction in I/O calls results in the response times that are 66-68% faster than the $PPHJ_p$ variants, similar to the previous experiment with unequal inputs. Table 10 gives the detailed data for the low contention situation. Our variants have 2-2.5 times the I/O volume of the $PPHJ_p$ variants, but only about one seventh as many I/O calls. This savings in the number of I/O calls results in response times that are 10-17% faster than the $PPHJ_p$ variants.

In summary, this experiment has demonstrated that our technique of dynamically adjusting the cluster size is very effective at all levels of contention, compared to previous techniques. Our

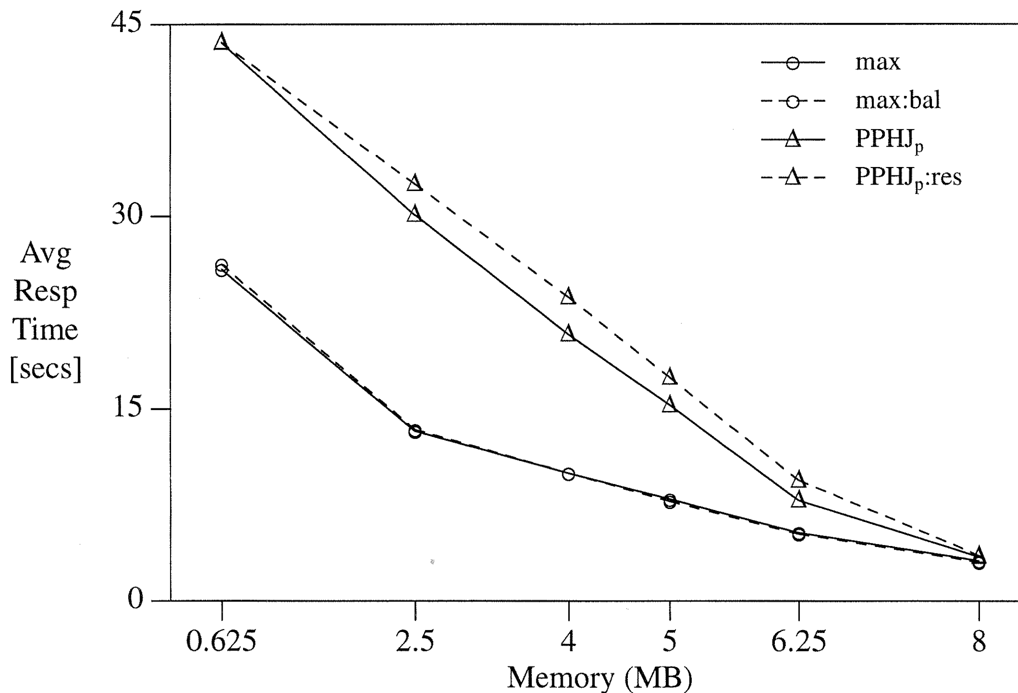


Figure 7. Varying Contention ($R = S = 5MB$).

| Alg. Variant | Average I/O | | | | Average Response Time [seconds] |
|---|---|---|---|---|---|
| | Total $IO_{calls}$ | Total $IO_{vol}$ [pages] | Build $IO_{vol}$ [pages] | Probe $IO_{vol}$ [pages] | |
| max | 1046 | 3262 | 1632 | 1630 | 25.87 |
| max:bal | 1082 | 3284 | 1666 | 1618 | 26.28 |
| $PPHJ_p$ | 3220 | 3220 | 1626 | 1594 | 43.58 |
| $PPHJ_p$:res | 3238 | 3238 | 1674 | 1564 | 43.60 |

Table 9. IOs for High Contention ($M = 0.625MB$, $R = S = 5MB$).

| Alg. Variant | Average I/O | | | | Average Response Time [seconds] |
|---|---|---|---|---|---|
| | Total $IO_{calls}$ | Total $IO_{vol}$ [pages] | Build $IO_{vol}$ [pages] | Probe $IO_{vol}$ [pages] | |
| max | 24 | 408 | 220 | 188 | 3.14 |
| max:bal | 27 | 374 | 274 | 100 | 3.02 |
| $PPHJ_p$ | 162 | 162 | 138 | 24 | 3.44 |
| $PPHJ_p$:res | 197 | 197 | 183 | 14 | 3.54 |

Table 10. IOs for Low Contention ($M = 8MB$, $R = S = 5MB$).

algorithm variants achieve significant reductions in response time compared to $PPHJ_p$ variants at higher levels of contention, and a smaller reduction at a very low level of contention. This is the case for inputs that differ in size significantly as well as for inputs that are equal in size.

## 6.3. Varying Frequency

In addition to being effective for large memory fluctuations and widely differing magnitudes of contention, an adaptable algorithm should also be stable with respect to extreme variations in the frequency of the memory fluctuations. The rate of memory fluctuations will vary depending on the amount of concurrent activity in the system. If there is much concurrent activity, the join may experience a high rate of fluctuations. However, if there is little concurrent activity, the join is more likely to experience a lower rate of fluctuations and retain its memory allocation for a longer duration. It is desirable for the join to provide consistent performance regardless of the frequency of the memory fluctuations. In this experiment, we examine the stability of the algorithm variants with respect to extreme variations in the frequency of the memory fluctuations. We vary the mean frequency of the fluctuations from 0.1 second to 10 seconds. The total memory from which the join's allocation is taken is $M = 2\ MB$, representing a medium-high contention environment. The join inputs for this experiment are $R = S = 5MB$. All other parameters are as described in the previous experiment in Section 6.2.

Figure 8 shows the response times of the algorithms as the fluctuations vary in frequency from very frequent to infrequent. To be effective, an adaptive technique must be faster than the fluctuation frequency. When fluctuations occur very frequently, $PPHJ_p$:res significantly increases the response time over $PPHJ_p$, as reported in [PCL93]. Restoration is an adaptive technique of coarse granularity. It is an expensive operation that takes a certain amount of time to complete once it is initiated, thus, its performance is affected by the rate of fluctuations. In contrast, our techniques provide a finer granularity of adaptation. As shown in Figure 8, *max* and *max:bal* are stable and provide consistent performance over the entire range of frequencies. Our techniques do not incur a large overhead and are, therefore, unaffected by the frequency of the fluctuations.

## 7. Summary and Conclusions

Dynamically varying the cluster size, or unit of I/O, is an effective technique for hash joins that experience fluctuations in their memory allocation during execution. Large clusters have been shown to be effective for algorithms with static memory allocations, but this study is the first to show how they might be exploited in a dynamic environment. While previous adaptable techniques focus on reducing the volume of I/O, our techniques reduce the amount of *time* spent on I/O.

We evaluated several algorithm variants that dynamically adjust the cluster size based on the current memory allocation, and one static variant that does not respond to the variations in memory availability. The static technique, *stat*, was unable to effectively exploit memory gains. The most effective dynamic variant was *max:bal*. This variant maximizes the size of each I/O request, and balances the use of additional memory between large clusters and restoration, a recently proposed technique. Restoration exploits memory increases during the probe stage by reading spilled build partitions back into memory to avoid probe I/O. *Max:bal* determines the

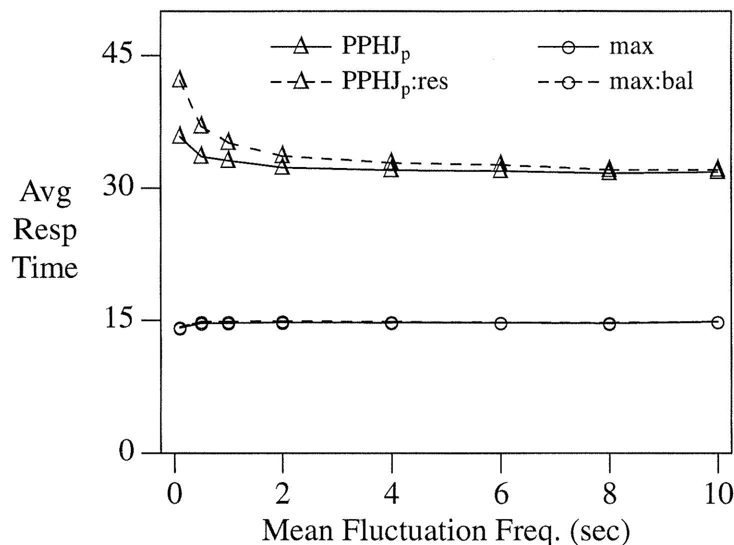


Figure 8. Varying Frequency of Fluctuations.

cluster size that would achieve most of the reduction in estimated response time, but it does not guarantee this amount to any partition. Memory in excess of what would be required to enlarge each spilled partition buffer to one cluster is used for restoration. Read requests are optimized by using a large input buffer. When a write is necessary, *max:bal* maximizes the size of the write request by writing the partition that has the largest amount of memory allocated. That is, if a resident partition must be spilled, *max:bal* spills the largest resident partition; if a spilled partition must be flushed, it flushes the spilled partition with the most pages assigned to its output buffer. Output buffers of spilled partitions may not exceed one cluster, but they are dynamically enlarged and reduced so that memory assigned to a partition is never idle. The output buffer of a partition that is spilled or flushed is reduced to one page, and may increase in size as needed, subject to memory availability. This dynamic resizing of spill buffers is a key contributor to the performance gains reported in this paper.

Our experimental evaluation included the Partially Preemptible Hash Join algorithm (PPHJ) that was recently shown to have better performance than earlier adaptable algorithms. PPHJ achieved its performance improvement by reducing I/O volume. However, our experimental evaluation showed that using large, dynamically sized clusters to increase I/O bandwidth allows the join to adapt much more effectively. Our techniques provide better performance for both large and frequent fluctuations, and for various magnitudes of contention. Moreover, the use of large clusters increases responsiveness, the ability to reduce quickly memory usage, which is important to overall system performance. We conclude that the current focus on reducing I/O volume to adapt to fluctutations in memory availability is misguided and should be refocused on decreasing I/O time, since it significantly improves both performance and responsiveness.

## Acknowledgements

We are grateful for helpful discussions with Rick Cole, Denise Ecklund, Leonard Shapiro, and Richard Wolniewicz. This research was partially supported by a grant from Digital Equipment Corporation. Diane L. Davison was partially supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland.

## References

[Bra84] K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations", *Proc. Int'l. Conf. on Very Large Data Bases*, Singapore, August 1984, 323.

[DKO84] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems", *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, 1.

[Gra90] G. Graefe, "Parallel External Sorting in Volcano", *Univ. of Colorado at Boulder Comp. Sci. Tech. Rep. 459*, 1990. Available via anonymous ftp from ftp.cs.pdx.edu as pub/faculty/graefe/papers/sort.ps.Z.

[Gra93a] G. Graefe, "Performance Enhancements for Hybrid Hash Join", *submitted for publication*, 1993.

[Gra93b] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys 25*, 2 (June 1993), 73-170.

[GLS94] G. Graefe, A. Linville, and L. D. Shapiro, "Sort versus Hash Revisited", *to appear in IEEE Trans. on Knowledge and Data Eng.*, 1994.

[IoC91] Y. E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results", *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 268.

[KTM83] M. Kitsuregawa, H. Tanaka, and T. Motooka, "Application of Hash to Data Base Machine and Its Architecture", *New Generation Computing 1*, 1 (1983), 63.

[KNT89] M. Kitsuregawa, M. Nakayama, and M. Takagi, "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method", *Proc. Int'l. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989, 257.

[NKT88] M. Nakayama, M. Kitsuregawa, and M. Takagi, "Hash-Partitioned Join Method Using Dynamic Destaging Strategy", *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 468.

[PCL93] H. Pang, M. J. Carey, and M. Livny, "Partially Preemptible Hash Joins", *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 59.

[Sal88] B. Salzberg, *File Structures: An Analytic Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Sha86] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Trans. on Database Sys. 11*, 3 (September 1986), 239.

[SSU91] A. Silberschatz, M. Stonebraker, and J. Ullman, "Database Systems: Achievements and Opportunities", *Comm. of the ACM, Special Section on Next-Generation Database Systems 34*, 10 (October 1991), 110.

[ZeG90] H. Zeller and J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments", *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 186.

# Table of Contents

Abstract ..... 1
1. Introduction ..... 1
2. Related Work ..... 2
3. Responsive Hash Joins ..... 5
Basic Algorithm Description ..... 6
Algorithm Discussion: Memory Fluctuations ..... 9
Algorithm Variants ..... 10
4. Other Adaptable Algorithms ..... 12
5. Database Simulator ..... 12
6. Experimental Results ..... 14
6.1. Poorly Predicted But Stable Memory Allocation ..... 15
6.2. Varying Contention ..... 18
6.3. Varying Frequency ..... 22
7. Summary and Conclusions ..... 23
Acknowledgements ..... 24
References ..... 24

# List of Figures

Figure 1. Memory-Contention Responsive Hash Join — Basic Algorithm ..... 6
Figure 2. Partitioning with Dynamic Clusters ..... 8
Figure 3. Cluster Size with Most of the Performance Gain ..... 11
Figure 4. Large Memory Increase ($1MB$ $to$ $4MB$) ..... 15
Figure 5. Large Memory Decrease ($4MB$ $to$ $1MB$) ..... 17
Figure 6. Varying Contention ($R = 5MB, S = 50MB$) ..... 19
Figure 7. Varying Contention ($R = S = 5MB$) ..... 21
Figure 8. Varying Frequency of Fluctuations ..... 23

# List of Tables

Table 1. Machine Architecture ..... 13
Table 2. Number of CPU Instructions per Operation ..... 13
Table 3. Algorithms and Options ..... 14
Table 4. IOs for Large Memory Increase ..... 16
Table 5. IOs for Large Memory Decrease ..... 17
Table 6. Algorithm Responsiveness ..... 18
Table 7. IOs for High Contention ($M = .625MB$, $R = 5MB$, $S = 50MB$) ..... 20
Table 8. IOs for Low Contention ($M = 8MB$, $R = 5MB$, $S = 50MB$) ..... 20
Table 9. IOs for High Contention ($M = 0.625MB$, $R = S = 5MB$) ..... 22
Table 10. IOs for Low Contention ($M = 8MB$, $R = S = 5MB$) ..... 22