Memory-Contention Responsive Hash Joins

Diane L. Davison
University of Colorado at Boulder
Department of Computer Science
davison @ cs.colorado.edu

Goetz Graefe
Portland State University
Computer Science Department
graefe @ cs.pdx.edu

Abstract

In order to maximize system performance in environments with fluctuating memory contention, memory-intensive algorithms such as hash join must gracefully adapt to variations in available memory. Mixed workloads, creating fluctuations of erratic frequency and magnitude, make responsiveness to memory contention particularly important. Previous studies on adaptable hash joins have focused on lowering I/O costs by reducing the I/O volume, as measured in the number of pages, by spilling partitions from memory to disk and then restoring them into memory if more memory becomes available. In this paper, we present *memory-contention responsive hash joins* that (i) reduce the amount of *time* spent on I/O by using large I/O buffers, or *clusters*, (ii) dynamically vary the cluster size in response to fluctuations in memory availability, and (iii) employ earlier techniques of dynamic destaging and restoration. Our simulation results demonstrate that these combined techniques provide better performance than previous algorithms, particularly in environments with medium to high memory contention or with very frequent changes in memory availability.

1. Introduction

Static memory allocation techniques are inadequate for query execution in a multi-user environment because the system is unable to predict when new requests will arrive. Differences in query complexity and size complicate allocation decisions 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 consistently achieving good performance using static memory allocation techniques since an optimal division of memory among competing queries may become sub-optimal very quickly as queries enter and leave the system. To overcome this uncertainty, the system must adapt to changes or fluctuations in memory contention by adjusting previous allocation decisions, and memory-intensive algorithms such as hash join and sort must have the capability to respond gracefully to changes in their memory allocation at any time *during* their execution.

In this paper, we show how memory can be used more effectively 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 varying the cluster size depending on memory availability, and maximizing the size of I/O requests.

There are two approaches to reducing I/O cost: reduce *I/O volume*, or reduce *I/O time*. I/O volume measures the number of pages transferred to and from disk, whereas I/O time measures the total time for disk seeks, rotational latencies, and transfers. Most previous hash join algorithms, static or adaptable, have taken the former approach. They attempt to achieve cost savings through reduction of I/O volume by keeping as much of the build input resident as possible, and to maximize memory utilization by creating partitions equal in size to the join's memory allocation. However, the more important issue is the I/O time. Optimizing I/O time involves a tradeoff between I/O volume and the number of I/O operations, because large clusters result in fewer disk accesses but on the other hand may increase the I/O volume since memory pages are committed to spilled partition buffers that otherwise would be used for resident partitions. For static memory allocations, analyses and experiments have shown that both sort and hash-based algorithms are much more effective using relatively large clusters or I/O buffers to reduce I/O time, even if it increases I/O volume [Bra84,

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

Gra93a, GLS94]. Large clusters reduce the number of I/O calls and thus reduce total seek time and rotational latency, the most expensive components of small I/O operations.

A large, fixed cluster size that remains unchanged for the duration of a hash join realizes the benefit of large I/Os, but it has two significant problems in environments with changing memory contention. First, it increases the join's minimum memory requirement since at least one cluster is required for each partition. Second, it prevents the join from exploiting increases in memory to reduce I/O time. Thus, 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. The present paper presents solutions for these two problems.

In Section 2, we review earlier research on hash join algorithms, focusing on adaptable algorithms. Section 3 introduces our "memory-contention responsive hash join algorithm," discusses how it responds to memory fluctuations, and describes some variations of the basic algorithm. We describe our simulator and simulation parameters in Section 4 before presenting our experimental results in Section 5. Finally, we summarize the paper and offer our conclusions.

2. Related Work

In this section, we review previous research related to our work. Before beginning this discussion, we introduce notation and terminology used throughout this paper.

The build input is designated R, the probe input S, and the number of pages in the input buffer I. 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. R, S, I, M, and C are also used to express the respective sizes. All sizes are measured in pages. The number of partitions into which the inputs are divided is called the *partitioning 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. To account for hash table overhead, we use a fudge factor *fudge*; thus, $\mathit{fudge} \times R$ pages of memory are required to keep the entire build input in a 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 and produces either pairs of partition files to be processed in later steps, join output tuples, or both. 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 $M_{\min} = F + I$ pages, and the join cannot reduce its memory consumption below this amount.

Early work on hash joins assumed fixed memory allocations for the duration of the join [Bra84, DKO84, DeG85, 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., $\mathit{fudge} \times R \leq F \times M$ [DKO84, Sha86]. If there is sufficient memory, hybrid hash join will keep one partition resident to reduce the I/O volume (each partition is composed of multiple hash buckets). Hybrid hash join statically assigns memory to partitions and predetermines which partitions will be spilled. The *bucket tuning* technique employed in the Grace hybrid hash join hashes tuples into a large number of buckets and then groups resident buckets into a single resident partition and spilled buckets into memory-sized partitions for subsequent steps [KNT89]. To counteract the effects of skew in the build input, the technique of *dynamic destaging* dynamically chooses the largest bucket to spill (destage) when memory is exhausted [NKT88].

Zeller and Gray were the first to propose a hash join that can adapt to variations in available memory [ZeG90]. They use bucket tuning, and also propose using large cluster sizes. However, the cluster size C (as well as F and the number of buckets) is provided by the optimizer and the paper does not elaborate on how it is determined. The algorithm performs only one level of partitioning and resorts to a hashed-loops algorithm [DeG85] for partitions that are larger than memory. To try to keep partitions smaller than memory, if a partition grows beyond the join's current memory allocation, the fan-out is dynamically increased by splitting the large partition into two smaller partitions. This means that pages associated with the original, larger partition must be read twice or more in later steps. 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 the build 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. Even though it uses large clusters, the algorithm essentially retains the same goals as earlier algorithms as it focuses on reducing I/O volume and maximizing memory usage by creating partitions no larger than memory.

More recently, Pang et al. proposed the partially preemptible hash join (PPHJ) that adapts to memory fluctuations during both the build and probe stages [PCL93a, PCL93b]. PPHJ performs a single level of partitioning and uses classic in-memory hash join to process spilled partitions. The algorithm assumes accurate estimates of input sizes; however, it can be modified to handle estimation inaccuracy. PPHJ uses a fixed output buffer size of $C = 1$ page and a fan-out of $F = \sqrt{\mathit{fudge} \times R}$ that results in partitions of average size $\sqrt{\mathit{fudge} \times R}$ pages.

Pang et al. consider the input buffer to be overhead rather than part of the join's memory allocation, so PPHJ has a minimum memory requirement of $M_{\min} = F$ pages. Partitions are spilled according to partition number from highest to lowest, so that partition F is spilled first, and partition 1 is spilled last. Excess memory allocated to the join is used as a local I/O buffer [Sha86]. Pages in the local buffer are prioritized by partition number, with pages from lower numbered partitions having higher priority. A decrease in memory during either the build or probe stage causes one or more build partitions to be spilled. The main contribution of the algorithm is the technique of exploiting memory gains during the probe stage to restore spilled build partitions to memory. This saves probe I/O at the expense of additional build I/O. Partitions are restored in the reverse order that they are spilled, so some pages may be obtained from the local buffer, avoiding disk I/O. Pang et al. performed a simulation study comparing PPHJ to previous techniques and found that restoration is very effective when the inputs differ in size. Restoration was found to harm performance when the join's memory allocation fluctuates very rapidly.

3. Memory-Contention Responsive Join

An adaptable algorithm must satisfy two goals. First, it must be capable of both *capitalizing* on *memory increases* and *gracefully degrading* in the face of *memory losses*. The algorithm 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, which measures how fast an algorithm can reduce its memory usage when requested to do so by the memory manager. Responsiveness is particularly important in systems with frequent fluctuations in contention or with occasional high-priority requests.

To accomplish these goals, our hash join algorithm relies on two principal techniques. First, it dynamically adjusts the cluster size of the partitioning output buffers and of the input buffer depending on memory availability. It exploits additional memory to reduce the I/O time, and improves responsiveness by increasing the size of both read and write requests when possible. Second, it maximizes the size of write requests by choosing a partition to write based on current memory occupancy. Specifically, if it is necessary to spill a resident partition, the largest resident partition will be chosen (as in dynamic destaging [NKT88]); if it is necessary to flush buffered pages of some spilled partition, the spilled partition with the largest current buffer will be flushed. This technique reduces I/O time and improves algorithm responsiveness as it maximizes the size of write requests and, therefore, the effective disk bandwidth.

Before describing our algorithm, it is important to briefly consider the difference between dynamic and fixed clusters. An algorithm with a fixed cluster size uses that cluster size as the *mandatory* unit of I/O, so that it is the unit of I/O for all I/O requests. Instead, we define a *target cluster size* C_{tgt} for I/O buffers. The actual cluster size, or unit of I/O, may dynamically vary from one page up to C_{tgt}. Thus, the goal is to use I/O buffers of C_{tgt} pages, but the ability to realize that goal depends on memory availability. The target cluster size C_{tgt} is predetermined based on the I/O subsystem's capabilities and performance characteristics. Figure 1 shows the I/O cost curve for the disk used in our experiments as the cluster size increases. To generate this graph, we calculated the approximate cost in seconds to perform one I/O operation of *numPages* = 32 pages, as *clusterSize* varied from 1 to 32 pages. The formula we used is *ioCost* = *(rotLatency* + *seekTime)* × *(numPages* / *clusterSize)* + *numPages* × *pageSize* / *transferRate*. The most effective target cluster size is the one that allows realization of most of the cost savings from a large cluster size. This is the point at which the cost curve begins to level out, and such a point is illustrated in Figure 1. A cluster size $C_{eff} \leq C_{tgt}$ that achieves a significant fraction of the cost savings from a large cluster size is also illustrated in Figure 1. This cluster size is the *minimum effective cluster size*, the smallest cluster size we would like to use (the purpose of this cluster size is discussed in the algorithm description). It may seem that the choices for C_{tgt} and C_{eff} are arbitrary, and there is indeed flexibility in choosing these values. In the experimental section, we show that the main consideration is to avoid very small cluster sizes. One other distinction between fixed and dynamic clusters is that tuples may span the pages of a fixed cluster, but not a dynamic cluster. This allows pages to be read with a different cluster size than the one with which they were written.

3.1. Basic MCRHJ Algorithm

The entire hash join is composed of a sequence of one or more steps; the pseudo-code in Figure 2 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 overflow partitions, ensuring that our algorithm does not depend on accurate estimates of the input sizes. While Zeller and Gray's

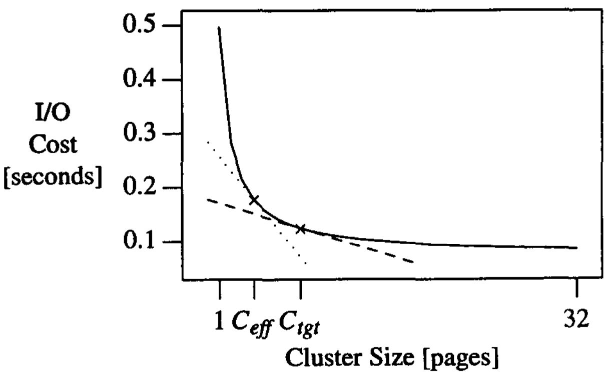


Figure 1. Target Cluster Size.

```
// step initialization
determine fan-out $F$ and minimal memory $M_{\text{min}}$
for each partition
  set state to resident
  obtain one page for a hash table
obtain up to $C_{tgt}$ pages for the input buffer
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's current pages
      insert the tuple into the partition
    else if a new page is available
      enlarge the partition with a new page[1]
    else if the partition is resident
      if there is a spilled partition with buffer $> C_{eff}$
        flush it and reduce its buffer to one page
      else
        spill the largest resident partition
        return all pages except one to the free list
    else // partition is spilled
      flush the spilled partition with the largest buffer
      return all pages except one to the free list

// 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 discard the tuple
  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[1]
      else
        flush the spilled partition with the largest buffer
        return all pages except one to the free list
    insert the tuple into the partition's output buffer
```

Figure 2. Memory-Contention Responsive Hash Join.

algorithm also handles inaccurate estimates of input sizes, their algorithm resorts to a hashed-loops algorithm if the initial partitioning step produces partitions larger than memory, whereas our algorithm uses full hash-partitioning as long as necessary. Techniques that prevent useless partitioning of inputs with very many duplicate join key values are considered elsewhere [Gra93b].

At the beginning of a step, it is necessary to determine the partitioning fan-out F. For the first step, the fan-out is set to $F = \sqrt{fudge \times R_{est}}$ as in PPHJ. Note that for extremely large R_{est}, we might want to limit F to some value that depends on the available machine, and thus presume multiple partitioning levels right from the start. Notice also that R_{est} is an estimate of the size of R; our algorithm does not depend on accurate estimates. For all later steps, F is set to $\sqrt{fudge \times R_i}$, where R_i pages is the actual size of the build partition file. F will be smaller for later steps since the partition files are smaller. 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 is similar to other adaptable algorithms [NKT88, PCL93a, ZeG90]. The input buffer is allocated as many pages as possible up to C_{tgt} pages. The hash join operator keeps a free list with any unused pages.

During the build stage, all build tuples are consumed and each is hashed to a partition, say P_i. A tuple that is assigned to a resident partition is copied into one of the partition's previously allocated pages and inserted into the hash table. If there is no space for the tuple in the partition's pages, a new page may be obtained and allocated to this partition. If no new page is available in the operator's free list, the algorithm will flush (and reduce to one page) a spilled partition that currently uses more than C_{eff} output buffer pages. Specifically, the largest such partition is chosen. If there is no such partition, the largest resident partition is spilled and reduced to a single page as an output buffer. The join spills the largest resident partition because this choice maximizes the write request and creates the most available memory. More importantly, spilling the largest resident partition is the most reasonable action if nothing is known about hash value skew in the probe input [NKT88] or if the hash value distribution in the probe input is uniform; better techniques are discussed in [Gra93b].

At this point, either P_i is still resident and may obtain an additional page, or P_i has been spilled. A tuple assigned to a spilled partition is copied to the partition's output buffer if there is space. If the buffer is full but smaller than C_{tgt} pages, the join will *enlarge the spilled partition buffer* by obtaining a new page. If no free page is available in the operator's free list, some pages are freed using the sequence of choices and actions above. The only difference is that, while resident partitions may be enlarged without limit, the buffers of a spilled partition are flushed before they can grow beyond the target cluster size C_{tgt}.

Let us consider an example. Figure 3 illustrates both minimal and enlarged spill buffers. The dashed box represents the target cluster size $C_{tgt} = 4$ pages for the output buffers of spilled partitions. In Figure 3, partition 1 has a minimal output buffer of 1 page, while the output buffer for partition 2 has been enlarged to the target cluster size of 4 pages. Thus, partition 1 may grow further, while partition 2 may not. Note that it would be possible to allow the spill buffers of partitions 1 and 2 to grow without limit, and to reclaim the pages when needed. However, there would likely be some limit placed on the maximum size of an I/O request, and no increase in the actual I/O size would be realized from allowing the spill buffers to exceed this limit (since the cluster would be written with multiple I/O

[1] If the partition is spilled, the size of its output buffer is limited to C_{tgt}. If the partition is resident, its hash table may grow without limit. See the discussion in the text.

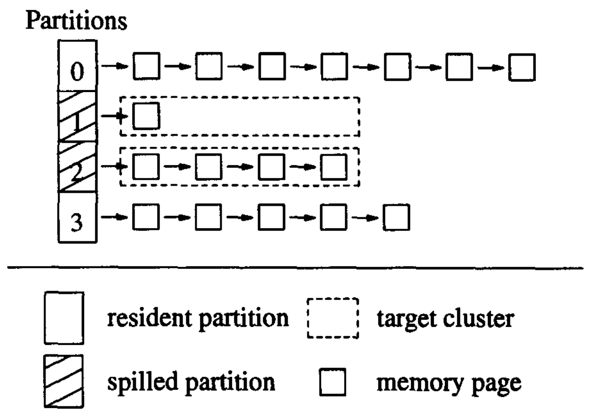


Figure 3. Partitioning with Dynamic Clusters.

requests anyway). Allowing the spill buffers of build partitions to grow without limit would result in an I/O savings in the case of restoration.

After all build input tuples have been consumed and all spilled build partitions have been flushed, 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 and smaller than C_{tgt} pages, the partition buffer will be enlarged with a new page if one is available; otherwise, the spilled partition with the largest output buffer is flushed. If a spilled partition's output buffer is full and equal to C_{tgt} pages, it is flushed and reduced to one page. This sequence of growing, flushing, and reducing the output buffer of a spilled partition 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.

3.2. Reaction to Memory Fluctuations

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. However, a fluctuation *during* a step requires special action.

Additional memory is utilized to enlarge the join's input and output buffers during both the build or probe stage. In response to an increase in memory, the input buffer is increased to C_{tgt} pages, if possible. A large input buffer is beneficial for all partitions, so it is a wise investment of memory. Any remaining pages are assigned to the join's free list so that they may be used to enlarge partition buffers as needed. For example, in Figure 3, additional memory could be allocated to any of partitions 0, 1, or 3. In contrast to the use of additional memory in our algorithm, the algorithms by Pang et al. and by 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 ($M_{\min} = F + I$ pages). If the join's reduced memory allocation is already equal to $M_{\min}$, 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, which can be done instantaneously. Second, it will reduce the buffers of spilled partitions to one page by flushing the full pages from partition output buffers. To maximize the size of write requests, spilled partitions are flushed in the order of decreasing numbers of currently allocated buffers so that partitions with *larger* current output buffers are spilled before partitions with smaller output buffers. This reduces I/O time and improves algorithm responsiveness. Third, the algorithm will spill resident partitions and reclaim all but one page to be used as an output buffer. As discussed for the build stage, resident partitions are spilled in the order of decreasing size. For example, given the situation in Figure 3, 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. 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. Table 1 summarizes the differences between our MCRHJ and PPHJ algorithms.

3.3. Algorithm Variants

Like most complex algorithms, our basic join algorithm includes some policy choices. Let us consider some alternatives for two of them.

- **fixed cluster (Fix*N*):** Large cluster sizes have been shown to be effective for query processing algorithms with static memory allocations [Bra84, Gra93a, Gra94, Sal90]; however, it is not clear how effective a large fixed cluster size is when an algorithm's memory allocation dynamically varies. To provide a basis for comparison with our dynamic techniques, this variant uses a fixed input buffer of 6 pages, and a fixed cluster size C_{fix} for the output buffers of spilled partitions. The fundamental algorithm is the same as our basic MCRHJ algorithm, except that the cluster size is fixed. The advantage of this variant is that the fixed cluster size guarantees that all I/O requests will use a large cluster size; the basic MCRHJ may in some cases use very small cluster sizes. Clearly,

Table 1. Algorithm Comparison.

Option/Event	PPHJ Algorithm	MCRHJ Algorithm
size of input buffer	6 pages	dynamically varies from 1 to C_{tgt} pages
size of output buffers	1 page each	each dynamically varies from 1 to C_{tgt} pages
hash table (HT) of resident partition P_i is full	enlarge P_i HT if free page else spill resident partition with largest index (spill P_{i+1} before P_i)	enlarge P_i HT if free page else flush spilled partition with buffer $> C_{eff}$ else spill largest resident partition
output buffer of spilled partition P_i is full	flush P_i	enlarge P_i buffer if free page else flush spilled partition with largest output buffer
memory gain during build stage	permit resident partitions to grow without limit excess to local spool buffer	enlarge input buffer to C_{tgt} pages permit resident partitions to grow without limit permit output buffers of spilled partitions to grow to C_{tgt} pages
spilled partition to restore	spilled partition with smallest index (restore P_i before P_{i+1})	smallest spilled partition
memory gain during probe stage	*res* – restore spilled partitions excess to local spool buffer	permit output buffers of spilled partitions to grow to C_{tgt} pages *bal* – output buffers to C_{eff} pages *bal* – restore spilled partitions with remainder
memory loss	flush local buffer in blocks of 6 pages spill resident partitions	flush & reduce output buffers of spilled partitions spill resident partitions

the disadvantage is that the fixed cluster size results in a larger minimum memory requirement of $M_{\min} = 6 + F \times C_{fix}$ pages. This algorithm is referred to as Fix*N*, where $N = C_{fix}$.

• **balance large cluster size with restoration (*bal*):** Since large clusters have been shown to be effective for static memory allocations, and since restoration has been shown to be effective in certain situations when the memory allocation fluctuates [PCL93a, PCL93b], it is natural to try to combine the benefits of both techniques. To do so, we provide an option that attempts to balance the two by adding restoration to our basic MCRHJ algorithm. The *bal* variant responds to an increase in memory during the probe stage as follows. It first attempts to realize a significant fraction 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 1 illustrates the minimum effective cluster size C_{eff}, which is the smallest cluster size we would like to use. The *bal* variant reduces large output buffers of spilled partitions to C_{eff} pages and preserves enough memory that smaller output buffers of spilled partitions can be enlarged to C_{eff} pages. Any remaining memory is used for restoration. MCRHJ restores spilled partitions based on size, from smallest to largest, which is more effective than PPHJ for skewed build inputs but has no effect for uniformly distributed hash values. The *bal* technique can be applied to either the basic MCRHJ algorithm or the Fix*N* variant.

4. Database Simulator

We implemented a simulator to study how to adapt most effectively to memory fluctuations in a dynamic environment, and to determine the effectiveness of our techniques compared to previous approaches. In this section, we describe first the simulated hardware and software architectures and then the implementation of earlier adaptable hash join algorithms in the simulator.

The simulated machine has a single CPU, two disks, a memory manager, and a query source. Table 2 summarizes the machine architecture, with the disk parameters taken from a Maxtor MXT-1240S. The disk services requests in first-come-first-served (FCFS) order. The disk access time is *disk access = seek time + rotational delay + transfer time*. The time to seek across t tracks is calculated as *seek time = seek factor* $\times \sqrt{t}$ [BiG88]. Partition files are located on one disk, and disk access time is minimized by allocating the partition files on nearby cylinders. The base input files are located on a second disk. We use synchronous I/O in this simulation.

The join we model is a primary key-foreign key join in which each probe tuple matches with exactly one build tuple. The hash values are assumed to follow a uniform distribution. The number of CPU instructions for each simulator operation is given in Table 3, reproduced from [PCL93a]. 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

Table 2. Machine Architecture.

Architecture Parameter	Value
CPU Speed	20 MIPS
Page Size	8 KB
Number of Disks	2
Disk Transfer Rate	3.5 MB/sec
Disk Rotational Latency	4.76 ms
Number of Cylinders per Disk	2368
Disk Cylinder Size	69 pages
Disk Seek Factor	0.000247

Table 3. Number of CPU Instructions per Operation.

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

join is active; as soon as that join completes, another one is started. Concurrent operations are reflected in the fluctuating memory allocations. A join that is allocated less memory than its minimum memory requirement is suspended until additional memory is available. The allocation of memory among algorithms is actually a policy decision of the memory manager, including whether a join should ever be given less memory than its minimum requirement. We do not investigate such policy decisions in this paper, so we simply suspend the join [PCL93a]. Memory allocated to a join is considered "pinned" in the buffer and managed entirely by the join operator without system intervention. The memory manager decreases or increases the join's memory allocation, and the join responds to these changes by unpinning or pinning pages. All joins in our simulator respond as soon as possible to memory fluctuations. For example, if more memory becomes available again while a join is in the process of reducing its memory usage, it will cease reduction.

Simulation of PPHJ

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 [PCL93a, PCL93b]. Therefore, it is sufficient to compare our algorithms only to PPHJ. We discussed PPHJ in detail in Section 2; in this section, we describe our implementation of PPHJ. We implemented PPHJ as faithfully as possible based on the available algorithm descriptions [PCL93a, PCL93b], with one small modification to make PPHJ adaptable even while processing overflow files. We refer to the modified algorithm as PP_r, since it recursively partitions its inputs.

For steps after the initial step, PPHJ uses classic in-memory hash join. Thus, it is impossible to spill a partition during those steps, because no suitable partitions are created. PP_r, our modified version, recursively partitions its inputs and processes partition files in exactly the same way as it processes the base inputs. This modification allows the algorithm to adapt to memory fluctuations during all steps, and also enables it to handle inaccurate estimates of input sizes.

Among the variants of PPHJ studied by Pang et al., the most effective variant combines late contraction, restoration, and priority spooling, of which restoration provides the largest performance improvement. PP_r includes both late contraction and priority spooling. Restoration was effective in some but not all situations analyzed, so we include PP_r both with and without restoration (*res* option). The unit of I/O for the experiments presented in [PCL93a] was one page. In [PCL93b], response time was improved significantly by flushing pages from the local spool buffer in blocks of 6 pages. Therefore, PP_r uses an input buffer of 6 pages and flushes pages from the local buffer in blocks of 6 pages at a time.

5. Experimental Evaluations

In this section, we compare our techniques to each other and to PP_r for a variety of workloads. To ease comparison with the original papers on PPHJ, we structured our simulation experiments similarly to those presented in [PCL93a]. Since the techniques proposed by Pang et al. and in this paper vary in their management of memory and partition files but do not affect the time to read base inputs or write the join output, the performance metric reported here is the average *join response time*, which combines the CPU time specific to the join and the I/O time to partition files. Join response time, as opposed to query response time, correctly reflects the effort specific to the join, whether the inputs come from file scans, index scans, or pipelines, and was used both by DeWitt et al. and by Shapiro to analyze the effectiveness of join algorithms [DKO84, Sha86]. Absolute differences in join response time fairly accurately reflect differences in query response time, whereas relative differences are different due to the effort required to obtain the join inputs and to dispose of the join output, which is the same for all algorithms considered here.

We report I/O activity as the average number of I/O requests and average I/O volume in pages. The I/O volume may be further divided into I/O to either build or probe partition files. For the MCRHJ variants, we use $C_{tgt} = 8$ and $C_{eff} = 4$.

In Section 5.1, we examine how well the algorithms adapt under different magnitudes of memory contention.

The experiment in Section 5.2 analyzes the algorithms' performance with respect to different frequencies of memory fluctuations. Finally, in Section 5.3, we examine the sensitivity of MCRHJ to C_{tgt} and the sensitivity of MCRHJ:bal to C_{eff}.

5.1. Memory Contention

In this experiment, we examine the effectiveness of the adaptable techniques when they experience different degrees of memory contention. We simulate a situation in which the adaptable join competes for memory with other operations 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 selected uniformly 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 intervals between changes in contention are chosen from an exponential distribution with a mean of 1 second.

High Contention

To represent an environment with fairly high contention, the amount of system memory is set to $M = 1$ MB. The join inputs for this experiment are $R = 5$ MB and $S = 50$ MB. We could have chosen a smaller system memory, but the Fix4 variant (MCRHJ with a fixed cluster size of 4 pages) has a high minimum memory requirement of almost 1 MB so a smaller system memory would have forced us to exclude that variant.

The average join response times for the various algorithms under high contention are shown in Figure 4. There are a number of interesting observations that we can make from this figure. Restoration is ineffective under high contention; neither the *res* nor the *bal* options produce noticeable differences in join response time. MCRHJ provides the lowest join response times of all the variants, demonstrating the effectiveness of our algorithm under high contention. Our Fix2 variant provides better performance than PP_r, even though PP_r flushes its local buffer in blocks of 6 pages. Fix4 gives the worst performance, even worse than Fix2 although its cluster size is twice as large.

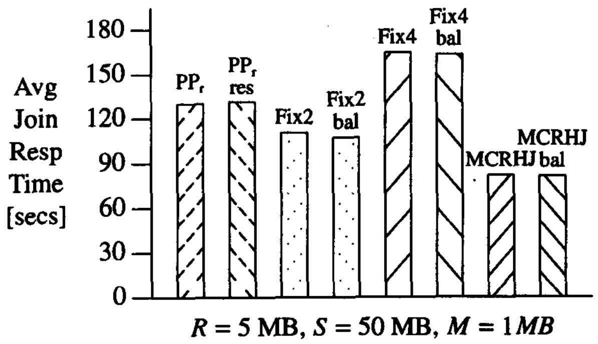


Figure 4. High Contention.

Table 4. Details for the High Contention Experiment.

Alg. Variant	Average I/O: Total IO_{calls}	Total IO_{vol} [pgs]	Build IO_{vol} [pgs]	Probe IO_{vol} [pgs]	Avg Resp Time [secs]
PP_r	9283	17483	1670	15813	129.99
PP_r:res	9429	17125	2475	14649	131.20
Fix2	5906	17568	1658	15910	110.86
Fix2:bal	5779	16981	2023	14958	107.58
Fix4	3703	17528	1622	15905	164.94
Fix4:bal	3696	17484	1635	15849	163.64
MCRHJ	2606	17517	1653	15864	82.09
MCRHJ:bal	2592	17352	1674	15678	81.42

To explain these observations, we examine the detailed data for the high contention situation, given in Table 4. The total I/O volume is similar for all algorithms; therefore, the significant differences must be due to I/O calls. MCRHJ has a significantly lower number of I/O calls than the other variants, with an *average actual cluster size* (*IOvol* / *IOcalls*) of 6.72. Thus, the dynamic cluster sizing techniques of MCRHJ very effectively permit large cluster sizes in this environment.

To understand why Fix2 outperforms PP_r, notice that Fix2 has a much lower number of I/O calls. Since both PP_r and the Fix*N* variants have an input buffer of 6 pages, the reason must be in differences in the output cluster sizes. Fix2 has an average actual cluster size of 2.97 compared to 1.88 for PP_r. Why does PP_r average only 1.88 pages per I/O call when it has an input buffer of 6 pages and attempts to flush the local buffer in blocks of 6 pages? The reason is that, since memory is scarce, PP_r does not often have excess memory to assign to the local buffer. Thus, most of its write I/O is not buffered but written directly using a cluster size of only one page.

Similarly, one might expect Fix4 to perform better than Fix2 since it has an output cluster size twice as large as Fix2. In fact, we see in Table 4 that Fix4 does indeed have a much lower number of I/O calls than Fix2; however, rather than giving better performance, it has the highest response time of all the variants. This is a result of its high minimum memory requirement of $M_{\min} = 6 + F \times 4$, which causes excessive suspension (recall that a join is suspended when its memory allocation falls below its minimum memory requirement). Despite achieving an average actual cluster size of 4.7 pages, more than PP_r and Fix2, Fix4's high minimum memory requirement is a serious impediment to good performance in a high contention environment.

In summary, this experiment has demonstrated that the techniques employed by MCRHJ are very effective in reducing response time in a high contention environment. This is accomplished by utilizing memory to reduce I/O time by dynamically maximizing the actual cluster size. Our techniques are effective compared to previous techniques as well as compared to large fixed cluster sizes, which cannot fully exploit memory gains and which may suffer excessive suspension as a result of a high minimum memory requirement. Furthermore, restoration cannot be exploited under high contention because there isn't sufficient memory to utilize the technique.

Low Contention

We now examine a low contention environment by repeating the previous experiment with the one difference that the system memory in increased to $M = 8$ MB to simulate a low level of memory contention. Given the experimental parameters with this large system memory, the join will have its maximum memory requirement of 6 MB (*fudge* × 5 MB) 85% of the time, so this experiment demonstrates *very* low contention[2].

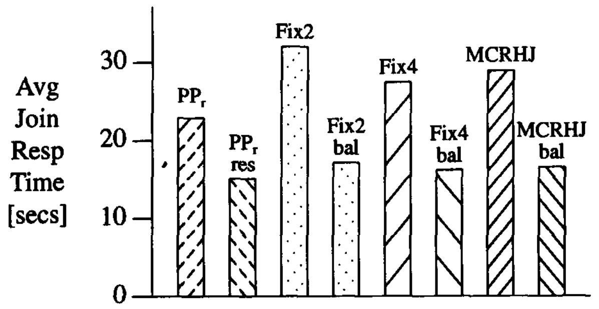
The average join response times for the various algorithms under low contention are shown in Figure 5. The most immediate contrast to the experiment with high contention is that restoration is quite effective for all of the algorithms. PP_r:res is the fastest algorithm here, closely followed by the three MCRHJ variants with restoration.

These differences are explained by the detailed data given in Table 5. Whereas memory scarcity under high contention prevents restoration, the abundance of memory here allows it to be exploited effectively. For example, PP_r:res uses about twice as much build I/O volume as PP_r (788 vs. 402 pages) to reduce the much larger probe I/O volume to about one tenth of PP_r (244 vs. 2307 pages), reducing the average join response time by 7.84 seconds. Restoration is also effective when combined with both MCRHJ and Fix*N* as the *bal* option, resulting in savings of 11.31 to 14.85 seconds. However, the join response times of MCRHJ:bal and Fix*N*:bal are a little higher than those of PP_r:res. Since their performance improvements are even larger than that achieved by PP_r:res over PP_r, the explanation must be due to fundamental differences in the basic algorithms. In Table 5, we can see that the total I/O volume for MCRHJ is about twice that of PP_r. The reason is that MCRHJ utilizes some memory to dynamically enlarge partition output buffers, making this memory unavailable for resident/restored partitions. Although this does effectively decrease the number of I/O calls, the penalty in increased I/O volume is higher than the benefit. The same situation occurs with Fix2 and Fix4, with the differences being due to their smaller actual average cluster size when compared to MCRHJ. The reader may have noticed that MCRHJ and MCRHJ:bal both have average actual cluster sizes larger than the target cluster size $C_{tgt} = 8$ (11.97 and 8.7, respectively). Although the input buffer and the partition output buffers are limited to C_{tgt} pages, the original I/O to spill a partition may use larger cluster sizes, and this is exactly what MCRHJ does. Since the majority of the write I/O volume for MCRHJ under low contention is original spilling, its average actual cluster size is quite large. For MCRHJ:bal, the cluster size for restoration is limited to C_{tgt}, so its average actual cluster size is smaller than that of MCRHJ.

As we had expected in the previous experiment, PP_r outperforms Fix2 which outperforms Fix4. Whereas under high contention, PP_r did not have sufficient memory to assign to its local buffer, it does under low contention. Thus, as opposed to the average actual cluster size of 1.88 pages per I/O calls under high contention, PP_r achieves an average of 3.23 under low contention. Of course, Fix2 (and Fix4) also have fixed output buffers, resulting in increased I/O volume compared to PP_r. Fix4 outperforms

Table 5. Details for the Low Contention Experiment.

Alg. Variant	Average I/O				Avg Resp Time [secs]
	Total IO_{calls}	Total IO_{vol} [pgs]	Build IO_{vol} [pgs]	Probe IO_{vol} [pgs]	
PP_r	837	2710	402	2307	22.87
PP_r:res	417	1033	788	244	15.03
Fix2	1439	4272	623	3648	31.96
Fix2:bal	572	1389	960	428	17.11
Fix4	867	4077	639	3438	27.43
Fix4:bal	343	1445	953	491	16.12
MCRHJ	456	5457	788	4668	28.94
MCRHJ:bal	218	1905	1274	630	16.54



$R = 5$ MB, $S = 50$ MB, $M = 8$ MB

Figure 5. Low Contention.

[2] The 85% is explained as follows: 80% of the time, the join will have ≥ 6.4 MB; of the remaining 20% of the time, the join is allocated ≥ 6 MB 25% of the time.

Fix2 as expected since it does not suffer from excessive suspension when there is ample memory.

To summarize, this experiment has shown that restoration (in the form of our *bal* technique) can be effectively combined with our dynamic cluster techniques to enhance the performance of the basic MCRHJ algorithm under very low contention. However, using some memory to enlarge I/O buffers increases the I/O volume, resulting in a slightly higher join response time for MCRHJ:bal than for PP_r:res.

Varying Contention

In this experiment, we subject the join algorithms to varying magnitudes of contention, ranging from high to very low contention. The previous two experiments demonstrated a significant advantage of MCRHJ:bal over PP_r:res at high contention, and a small advantage of PP_r:res over MCRHJ:bal at low contention. Now we examine the entire range in between to determine the effectiveness of the algorithms in a larger context. In this experiment, the system memory ranges from 1 MB (high contention) to 8 MB (very low contention). The join inputs for this experiment are $R = 5$ MB and $S = 50$ MB. Based on the performance in the previous experiments, we eliminated Fix4, Fix4:bal, and Fix2 since they provided inferior performance.

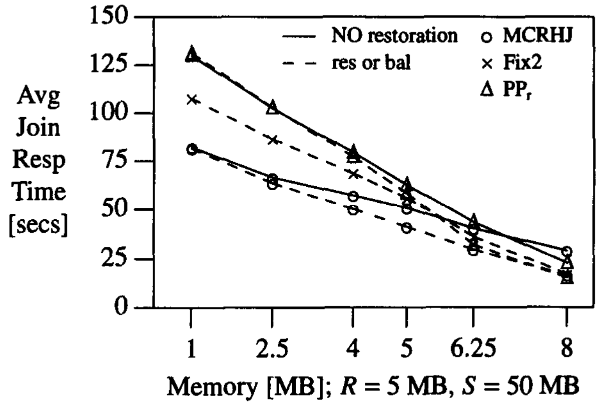


Figure 6. Varying Contention.

The average join response times for the various algorithms when contention varies from high to very low are shown in Figure 6. (the end-point values at 1 MB and 8 MB are taken from the previous two experiments). Figure 6 clearly shows the effectiveness of our MCRHJ algorithm over a wide range of memory contention. Obviously, our techniques provide much greater performance differences at higher levels of contention, but MCRHJ:bal provides the lowest join response times of all algorithms for all data points with memory size $M \leq 6.25$ MB. It is interesting that Fix2:bal outperforms PP_r:res over a significant range of contention (data points ≤ 5 MB), since PP_r:res has insufficient memory to write and read in large clusters (as discussed above for the high contention).

We also explored the sensitivity of the results to the relative input sizes, i.e., the ratio R / S. The results were very similar to those in Figure 6, so we do not include them here.

In summary, this experiment has demonstrated that our dynamic cluster techniques combined with restoration (MCRHJ:bal) are very effective over a wide range of levels of memory contention, compared to fixed cluster techniques and previous techniques that utilize only restoration. The only exception to this observation is at very low levels of contention when the total I/O volume is small; in this case, PP_r:res performed slightly better than MCRHJ:bal.

5.2. Interval between New Allocations

In addition to being effective for 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. 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 duration of an allocation from 0.1 second to 10 seconds, using an exponential distribution as in the previous experiment. The total memory from which the join's allocation is taken is $M = 3$ MB, representing a medium contention environment. The join inputs for this experiment are $R = S = 5$ MB. All other parameters are as described in the previous subsection.

To be effective, an adaptive technique must adapt faster than the fluctuation frequency. When fluctuations occur very frequently, restoration increases the response time because it is an adaptive technique of coarse granularity. That is, restoration is an expensive operation that provides no guaranteed benefit. Figure 7 shows the join response times of the algorithms as the fluctuations vary from

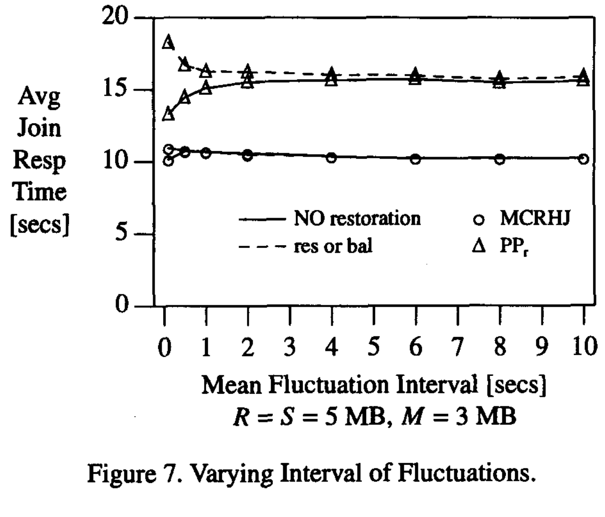


Figure 7. Varying Interval of Fluctuations.

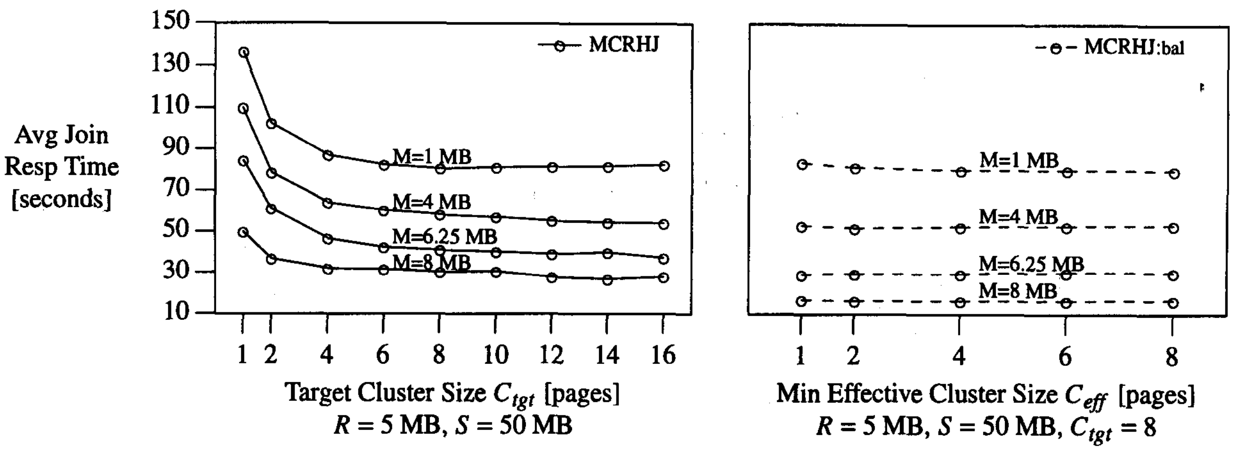


Figure 8. Sensitivity of MCRHJ to its Parameters.

frequent to infrequent. PP_r:res does indeed perform worse under frequent fluctuations because it suffers from the cost of restoration but does not have enough time to realize the benefit. MCRHJ:bal suffers very slightly from this, but not nearly as much as PP_r:res since MCRHJ:bal does not use all of its memory for restoration.

It is rather surprising to see that PP_r and MCRHJ actually achieve lower join response times at high fluctuation frequency. There are two reasons for this phenomenon. First, recall that our joins respond as soon as possible to changes in memory contention. When contention changes very frequently, both algorithms abort memory reduction efforts before they are finished, resulting in an I/O savings compared to less frequent fluctuations. Second, PP_r employs priority spooling, which is more effective at a high frequency of fluctuations. The total I/O volume for PP_r is very stable regardless of the fluctuation frequency, but at high frequency, it is able to achieve larger block writes resulting in a lower I/O calls.

In summary, our algorithm variants are more stable over a wide range of fluctuation frequencies, including very frequent fluctuations. Our techniques for exploiting additional memory are of a finer granularity than previous techniques. Even when combined with restoration, this expensive operation does not dominate since it is secondary to our dynamic cluster techniques.

5.3. Sensitivity Analysis

The two parameters that control our MCRHJ algorithm are the target cluster size C_{tgt} and the minimum effective cluster size C_{eff}. In this experiment, we examine the sensitivity of the MCRHJ algorithm to these parameters. Figure 8 shows the sensitivity of MCRHJ to both of these parameters. The left graph in Figure 8 shows the sensitivity to C_{tgt} for several different levels of contention, realized by different memory sizes M as in the previous experiments. A significant performance improvement can be obtained by simply avoiding very small target cluster sizes less than four. Not surprisingly, extremely poor performance results from a target cluster size of one. Figure 8 shows that our choice of $C_{tgt} = 8$ is quite reasonable. Using a target cluster size of C_{tgt} = 8, the right graph of Figure 8 shows the sensitivity of MCRHJ:bal to the choice of C_{eff} for different levels of contention. At lower contention, the algorithm is insensitive to this parameter; the minimum effective cluster size is not a limiting factor since there is ample memory and buffer sizes are likely already larger than C_{eff} (up to C_{tgt}). Under higher contention, particularly for M = 1 MB, a smaller C_{eff} does slightly worsen performance until it stabilizes at C_{eff} = 4. Thus, it is more important to attempt to realize some minimal cluster size when memory is scarce than when memory is ample, but a larger C_{eff} does not affect performance at lower contention.

In summary, it is quite simple to find effective parameters for C_{tgt} and C_{eff}. The main consideration is to avoid very small cluster sizes, which is easy to do using the I/O cost curve of Figure 1.

6. Summary and Conclusions

In this paper, we considered hash join algorithms that can adapt to changes or fluctuations in their memory allocation at any time during their execution. We showed the importance of reducing the amount of *time* spent on I/O, rather than only reducing the I/O volume, or number of pages of I/O. The two techniques we proposed to allow effective adaptation to both increases and decreases in memory allocation are (i) dynamically varying the size of the *clusters*, or I/O buffers, depending on memory availability, and (ii) maximizing the cluster size of I/O requests. Previous research has demonstrated the effectiveness of the techniques of dynamic destaging, which dynamically chooses a partition to spill based on size, and restoration, which uses additional memory to restore spilled build partitions to memory. Our new hash join algorithm, Memory-Contention Responsive Hash Join (MCRHJ), combines our two new techniques with the previous ones.

The new algorithm's most effective variant explored here is MCRHJ:bal. This variant defines a target cluster size of C_{tgt} pages, which is the I/O buffer size that achieves most of the cost savings from large I/O requests for a given I/O subsystem. The I/O buffers of spilled partitions may dynamically vary from 1 to C_{tgt} pages, depending on memory availability. When memory is plentiful, spilled partition buffers may be enlarged up to C_{tgt} pages, but in times of memory scarcity, one or more of them may be reduced to fewer than C_{tgt} pages. If more memory is available than is required to enlarge all spilled partition buffers to C_{eff} pages, it is used for restoring spilled build partitions to memory (C_{eff} is the cluster size $\leq C_{tgt}$ that achieves a significant fraction of the cost savings from using a large cluster size). Read requests are optimized by using a large input buffer. Write requests are optimized by writing the partition that has the largest amount of memory allocated. In other words, if a resident partition must be spilled, MCRHJ: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. The output buffer of a partition that is spilled or flushed is reduced to one page, and may then increase in size up to C_{tgt} pages, depending on need and memory availability.

Our experimental evaluation included the Partially Preemptible Hash Join algorithm (PPHJ) that was recently shown to have better performance than earlier adaptable algorithms. PPHJ achieves its performance improvement by using restoration to reduce I/O volume. However, our experimental evaluation demonstrates that large, dynamically-sized clusters allow the join to adapt more effectively at higher levels of contention. Our MCRHJ:bal variant achieves better performance than PPHJ at all levels of memory contention considered here, except at very low contention, and it achieves more stable performance for very frequent fluctuations. Moreover, using large clusters increases responsiveness, the ability to reduce memory usage quickly, thus permitting faster reaction to new high-priority queries entering a database system. We conclude that combining dynamically-sized I/O clusters, maximized I/O requests, dynamic destaging, and restoration results in the most effective join algorithm to-date for environments with fluctuating memory contention.

Acknowledgements

This research was partially supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland, Digital Equipment Corp., and Texas Instruments.

References

[BiG88] D. Bitton and J. Gray, "Disk Shadowing", *Proc. Int'l. Conf. on Very Large Data Bases*, Los Angeles, CA, August 1988, 331.

[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.

[DeG85] D. J. DeWitt and R. H. Gerber, "Multiprocessor Hash-Based Join Algorithms", *Proc. Int'l. Conf. on Very Large Data Bases*, Stockholm, Sweden, August 1985, 151.

[Gra93a] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys 25*, 2 (June 1993), 73-170.

[Gra93b] G. Graefe, "A Performance Evaluation of Histogram-Driven Recursive Hybrid Hash Join", *submitted for publication*, August 1993.

[Gra94] G. Graefe, "Sort-Merge-Join: An Idea whose Time has(h) Passed?", *Proc. IEEE Int'l. Conf. on Data Eng.*, Houston, TX, February 1994, 406.

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

[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.

[PCL93a] H. Pang, M. J. Carey, and M. Livny, "Partially Preemptible Hash Joins", *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 59.

[PCL93b] H. Pang, M. Carey, and M. Livny, "Partially Preemptible Hash Joins", *Univ. of Wisconsin – Madison Comp. Sci. Tech. Rep. 1144*, 1993.

[Sal90] B. Salzberg, "Merging Sorted Runs Using Large Main Memory", *Acta Informatica 27* (1990), 195, Springer.

[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.