Towards Eliminating Random I/O in Hash Joins [*]

Ming-Ling Lo and Chinya V. Ravishankar
Electrical Engineering and Computer Science Department
University of Michigan–Ann Arbor
1301 Beal Avenue, Ann Arbor, MI 48109
mingling, ravi@eecs.umich.edu

Abstract

The widening performance gap between CPU and disk is significant for hash join performance. Most current hash join methods try to reduce the volume of data transferred between memory and disk. In this paper, we try to reduce hash-join times by reducing random I/O. We study how current algorithms incur random I/O, and propose a new hash join method, Seq$^+$, *that converts much of the random I/O to sequential I/O.* Seq$^+$ *uses a new organization for hash buckets on disk, and larger input and output buffer sizes. We introduce the technique of batch writes to reduce the bucket-write cost, and the concepts of write- and read-groups of hash buckets to reduce the bucket-read cost. We derive a cost model for our method, and present formulas for choosing various algorithm parameters, including input and output buffer sizes. Our performance study shows that the new hash join method performs many times better than current algorithms under various environments. Since our cost functions under-estimate the cost of current algorithms and over-estimate the cost of* Seq$^+$*, the actual performance gain of* Seq$^+$ *is likely to be even greater.*

1 Introduction

Since the introduction of the GRACE hash join method [1], many other hash join algorithms have been proposed [2, 3, 4, 5, 6, 7]. Such work has focussed mainly on three goals: (1) reducing CPU costs, (2) reducing the amount of data transferred between memory and disk, and (3) improving the stability of hash joins in the presence of data skew.

However, machine capabilities have changed dramatically since the introduction of hash joins. CPU speeds and the sizes of memory and disk have all increased greatly, but increase in disk speeds has not matched. Another important factor is the significant disparity between the efficiency of random and sequential disk accesses [8]. As these trends continue, it is likely that I/O costs will dominate hash join costs even more. Minimizing I/O costs thus becomes crucial to join performance.

1.1 Sequential versus Random I/O

Hash-join methods have tried to improve performance by reducing the data transfer volumes between memory and disk. However, reducing data transfer volume is not the same as reducing I/O costs, since the costs of random and sequential I/O are very different. Reading ten disk blocks from disk in one sequential access generally costs much less than reading five blocks from disk in five random accesses. We argue that a good hash-join algorithm must transfer data between memory and disk in sequential I/O as far as possible. [9] explored using sequential I/O in the context of hybrid join.

In this paper, we propose a new hash join method that reads/writes its source, intermediate, and result data mostly using sequential disk accesses. To transfer intermediate data using sequential I/O operations, we introduce a new organization of intermediate data on disk, storing each bucket as a small number of bucket *segments* instead of one contiguous file. To support this disk layout, we use a technique called *batch writes*, which writes pages from multiple buckets to disk when the buffer becomes full, and enables buckets output mostly in sequential writes. We have used a similar technique to build seeded trees for spatial databases [10]. The new bucket organization on disk leads to the concepts of *write groups* and *read groups* for hash buckets, which significantly reduce bucket read costs with slightly increased bucket write cost. Transferring source and result relations with sequential I/O is conceptually straightforward, and is achieved by increasing the input and output buffer sizes.

Our method has the following features. First, using fixed or target I/O *clusters* either increases minimum memory requirement and/or limit the size a (spilled)

[*]This work was supported in part by the Consortium for International Earth Science Information Networking.

a bucket can grown before written to disk [11]. The concept of I/O cluster is immaterial to our method. Second, when writing bucket contents to disk during the partition phase, our method writes more than one bucket in one sequential write, while most earlier methods restrict their write sizes to less than one bucket. Like the dynamic hash GRACE join [5], our method allows a large number of hash buckets, and dynamically combines multiple buckets to form a hash table. Thus, it retains stability against bucket overflow similar to that of the dynamic hash GRACE join. Also, our work also discusses the I/O of source/result relations, which most current work ignores. We suggest a minimum amount for buffers for I/O.

This paper is organized as follows. Section 2 discuss the design of current hash joins and analyzes their costs. Section 3 describes our new method for hash join. Section 4 compares of the I/O costs for our method and other current methods, and Section 5 concludes this paper.

2 Previous Work

Traditional hash join algorithms have been discussed extensively in literature [2, 3, 4, 5, 6, 7]. Here we summarize only their relevant aspects. Given two relations to be joined, we assume without loss of generality that the smaller relation is the *inner relation*, denoted R, and the other relation is *outer relation*, denoted S. The result relation of the join is RES. The size of a relation U is denoted by $|U|$, the size of memory by M, and the size of a buffer B by M_B.

Hash join algorithms are generally divided into a *partition phase* and a *join phase*. In the partition phase, the source relations R and S are each partitioned into H_s disjoint *buckets* using the same hash function ϕ_1, called the *partition function.* Since source relations are generally larger than the buffer size, the total bucket size may become larger than the buffer size during partition, and bucket contents must be written to disk. In the join phase, each pair of inner relation/outer relation buckets is processed in turn. The inner relation bucket is read back into memory and built into a hash table using another hash function ϕ_2. The corresponding outer relation bucket is read into memory, its tuples hashed with ϕ_2, and probed against the hash table for R tuples with the same join attribute values.

Typically, the memory buffer is divided into the input buffer IB, the output buffer OB, each with one page, and the staging buffer SB, with the rest of the memory space. The input buffer is used to read both source relations during the partition phase, and to read the outer relation during the join phase. The output buffer is used to hold the result tuples before they are written to disk. The staging buffer holds bucket contents in the partition phase, and hash tables in the join phase.

The pages of the staging buffer are assigned to various buckets as necessary during the partition phase. We call the staging buffer space assigned to one bucket a *bucket buffer.* Each bucket buffer is at least one page in size. Since the number of SB pages is generally greater than the number of buckets, there are usually several pages per bucket buffer. Researchers have investigated various way for using the additional SB space.

The Hybrid Hash-join (HH join for short) method [2, 3] is reduces I/O costs by saving part of the inner relation in this additional space and never writing it to disk. The size of one of the buckets is taken to equal to the space not used for bucket buffers, and all its tuples are stored in this space through the partition phase. We call this bucket the *anchor bucket.* After the inner relation is partitioned, the anchor bucket is built into a hash table. As the outer relation is partitioned, tuples hashing into the anchor bucket are matched against the hash table immediately, producing some result tuples. Other tuples are inserted into their respective bucket buffers in SB. In the join phase, non-anchor buckets are processed as usual. HH maximizes bucket sizes to minimize their number and to save space for the anchor bucket. The anchor bucket is of size $M_{SB} - H_s + 1$, and all other buckets are of size M_{SB}. HH has the drawback that it adjusts the number of buckets so that the projected bucket sizes are approximately the size of available memory, and is thus most likely to have overflow buckets.

The Dynamic-Hashing GRACE Hash-join (DHGH join for short) [4, 5] tries to avoid overflow buckets, while retaining many of the benefits of the anchor bucket. During the partition phase, the DHGH method dynamically selects buckets for paging out to disk. It favors paging out buckets that have been written to disk before. After the inner relation is fully partitioned, the R buckets that have never been output to disk are built into one hash table. By analogy with HH join, we call such buckets anchor buckets. During the join phase, multiple R buckets can be combined to into one hash table, so that there are fewer hash tables and lower hash table initialization overhead. Since the hash table are assembled from multiple buckets, and the average buffer size is smaller, buckets overflow is much less likely.

2.1 Cost Analysis of Current Methods

Most current methods attempt to reduce the volume of data transferred between memory and disk, but not the number of random I/O operations. The cost analyses presented for these methods do not distinguish between random and sequential disk I/O, for the most part. Moreover, their design makes sequential I/O difficult.

We define the cost of a *random block access* to be the average cost of accessing a disk block when the disk head is at a random position on the disk. The cost of a random block access is thus the sum of average disk seek time, average rotation time, and the transmission time for one disk block. The cost of a *sequential block access* is defined to be that of accessing a disk block when the disk head is positioned at the beginning of that disk block. The term *sequential access* is used to suggest access to a series of adjacent disk blocks in a single disk I/O operation. Therefore, the cost a sequential read of ten contiguous disk blocks equals the cost of one random block access and nine sequential block accesses.

The I/O tasks of a hash join can be divided into four logical components: (1) Partition-phase reads (PR), (2) Partition-phase writes (PW), (3) Join-phase reads (JR), and (4) Join-phase writes (JW). We refer to tasks PR and JW collectively as *source/result I/O*, and to tasks of PW and JR as *bucket I/O*.

To the best of our knowledge, cost analyses for earlier methods did not discuss the cost of source/result I/O, since they are the same for all methods in terms of data transfer volume. The cost analysis in [2] is the only one to distinguish between random and sequential I/O, and treats PW as involving random writes and JR as involving sequential reads.

2.1.1 Partition-Phase Writes

Partition phase writes were listed in [2] as random writes. We attribute this to two reasons. First, since each bucket must be stored as a contiguous file on disk, it is necessary to seek to the end of the corresponding file whenever a bucket buffer is paged out. Second, only one output buffer page was used for each bucket, which must be written to disk when full. The order in which the bucket output buffers become full is random. Therefore, partition phase writes incur a series of random block writes.

2.1.2 Join-Phase Reads

Join phase reads were listed in [2] as sequential reads. Since a bucket is stored as a contiguous file in disk, it can be read back into memory with one large sequential read provided there is enough buffer space. This is true for the inner relation buckets, which are loaded into SB. However, it is unrealistic to treat the reads of outer relation buckets as sequential, for the following reasons:

[A:] There is only one IB page, which means the disk controller must load all blocks in a sequential read into the same buffer page.

[B:] Result tuples are produced as the S tuples are read. Since there is only one OB page, writes of result tuples occur frequently. The system must perform sequential reads of the S tuples and the frequent writes of result tuples currently, without interference between them.

[C:] Input S tuples are consumed as they are hashed and matched against the hash table. S tuples must be consumed before newly read blocks of S tuples overwrite old ones. However, the processing time for each S tuple depends on the number of R tuples it matches. If many S tuples with high match ratios are processed back to back, it is possible for the CPU to become a temporary bottleneck. More seriously, if the single output buffer page is full, input tuples will not be consumed till the output buffer is written to disk.

Overcoming difficulties A and B requires special hardware and operating system support. C is workload dependent and cannot be overcome with advanced system's support. We believe it is hard to avoid such difficulties, and earlier work has never actually indicated how to overcome them.

2.1.3 Partition-Phase Reads and Join-Phase Writes

The PR and JW tasks were not discussed in any earlier work we surveyed. PR reads the source relations and raises difficulties similar to A and B of join-phase reads: there is only one input buffer page for the source relations, and reading the source relations must proceed in parallel with the output of filled bucket pages.

JW outputs result tuples to disk. It is even more difficult to perform these writes as sequential I/O operations because all three difficulties of JR arise here. There is only one page output buffer for the result relation, the output proceeds in parallel with the input of outer relation tuples, and the rate at which result tuples are generated varies with the match ratio of incoming S tuples.

Our analysis suggests that for these hash join algorithms, PR must be performed as random accesses unless special hardware/OS support is available, while JW must be performed as random accesses in all cases.

3 Eliminating Random I/O

We consider random accesses in source/result I/O and that of bucket I/O separately. Current methods incur many random seeks during source/result I/O mainly because their input and output buffers are small, so one can reduce random I/O by simply increasing the sizes of IB and OB. The key issue we focus on is determining how large they can be made.

To reduce bucket I/O, our method stores each bucket on disk as a small number of bucket *segments* instead of as one contiguous file. Each bucket segment consists of a number of contiguous disk blocks, but different segments of the same bucket need not be adjacent. With this new organization, both partition phase writes and join phase reads are performed with sequential disk accesses of medium data size. We introduce the technique of ***batch writes*** to realize this disk layout for buckets. The idea of bucket segments also leads to the concepts of *write groups* and *read groups* of buckets, which significantly reduce the cost of reading in buckets with a slight increase in bucket write cost.

To simplify our discussions, we assume no overflow buckets. When bucket overflow occurs, our method recursively partitions the buckets into smaller sub-buckets till no sub-bucket overflows SB, as do traditional methods. We also assume there are H_s buckets, and $H_s \leq M_{SB}$. The cost of one random block access is denoted as A_{ran}, that of one sequential block access, A_{seq}. The ratio A_{ran}/A_{seq} is denoted as ρ. Other notations are the same as in Section 2.

3.1 Behavior of Input and Output Buffers

We read the source relations in chunks of M_{IB} pages, and write the result relations in chunks of M_{OB} pages. The cost of PR and JW can be shown to be:

$$C(PR) = (|R| + |S|)\frac{M_{IB} + \rho - 1}{M_{IB}}A_{seq}, \quad (1)$$

$$C(JW) = |RES|\frac{M_{OB} + \rho - 1}{M_{OB}}A_{seq}. \quad (2)$$

The formulae show that as M_{IB} and M_{OB} increase, $C(PR)$ and $C(JW)$ decrease. However, as the input and output buffers grow beyond a certain point, these decreases become marginal. As a rule of thumb, we suggest allocating approximately $\rho - 1$ to both the input and output buffers, and allocating more pages to them when memory is abundant. For modern computers, memory sizes are orders of magnitude larger than ρ, so this increase in input and output buffer sizes is unlikely to represent a serious demand on memory.

3.2 Batch Writes

To reduce random desk accesses in bucket I/O, we introduce the concept of *batch writes*. Under this scheme, the SB pages are initially put in a free page pool. When a tuple is hashed into a bucket with insufficient space, a new page is allocated from the pool and linked to the bucket.

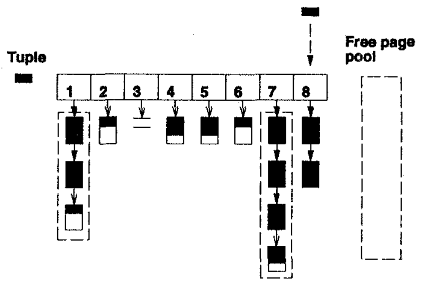

Figure 1: **A tuple awaits insertion, and the free page pool is empty. A batch write is executed using buckets 1 and 7.**

When a free page is requested and the free page pool is empty, we perform a batch write to reclaim part of the SB space. This is done by choosing a number of buckets and writing their pages to disk in one large sequential write (see figure 1).

We call the part of a bucket written to disk by a batch write a bucket *segment*. A batch write places the pages of a bucket segment in contiguous blocks on disk. However, different segments of the same bucket may reside in different locations on disk. During the join phase, we read a bucket into memory with one sequential read per segment. In general, the number of segments in a bucket is less than the number of batch writes, since a bucket need not participate in every batch write.

Two issues are important to the implementation of batch writes: (1) How much space should we reclaim in each batch write? (2) How should we select buckets to participate in batch writes? The next two subsections address these issues and derive the cost functions for PW and JR.

3.2.1 Partition-Phase Write Cost

Let us consider the first issue. Suppose we reclaim at least M_{bw} pages of memory with each batch write. The cost of PW is bounded from above by:

$$C(PW) \leq (|R| + |S|)\frac{M_{bw} + \rho - 1}{M_{bw}}A_{seq}.$$

This formula is the same as Equation 1 for $C(PR)$ if M_{IB} is substituted for M_{bw}. By similar arguments, we should reclaim at least $\rho - 1$ pages in each batch write for PW to be efficient. Reclaiming more than this amount provides additional but diminishing reductions in PW cost.

3.2.2 Join Phase Read Cost

Buckets of the inner and outer relations are read into different buffers. Thus batch writes affect their read costs differently. An inner relation bucket segment can be read sequentially if it is no larger than SB. By the way bucket segments are constructed, all bucket segments must be smaller than SB. Denote the average number of segments per bucket for relation U by $n_s(U)$, and the cost of reading relation R during the join phase by $C(JR_R)$. We have

$$C(JR_R) \quad = \quad (|R| + H_s n_s(R)(\rho - 1))A_{seq}.$$

There is no guarantee that all outer relation segments will be smaller than the input buffer. A segment is larger than IB must be read in chunks of size M_{IB}. Each segment thus yields some number of fixed-size chunks of size M_{IB}, and one residual chunk of size between 1 and M_{IB}. Given the total segment size and the number of segments, the number of fixed-size chunks is maximal when the residual chunks occupy the least space, which happens when all except one residual chunk is 1 block in size. The number of fixed-size chunks is thus bounded by $\frac{|S|-H_s n_s(S)}{M_{IB}}$. Recall each segment also contributes an additional residual chunk. Thus $n_c(S)$, the number of chunks in which the S segments are read, has the following upper bound: $n_c(S) \leq H_s n_s(S) + \frac{|S|-H_s n_s(S)}{M_{IB}}$.

Denoting the cost of reading relation S during the join phase by $C(JR_S)$, we have

$$C(JR_S) \leq (|S| + (H_s n_s(S) + \frac{|S| - H_s n_s(S)}{M_{IB}})(\rho - 1))A_{seq}.$$

The total cost of JR, $C(JR)$, is the sum of $C(JR_R)$ and $C(JR_S)$. Within $C(JR_R)$, the component $|R|A_{seq}$ corresponds to the time to actually transfer the buckets, while the component $H_s n_s(R)(\rho - 1)A_{seq}$ corresponds to the cost of disk head seeks to the starting positions. $C(JR_S)$ can be analyzed similarly. Depending on H_S and ρ, the cost of disk head seeks can dominate the join time read cost if not carefully controlled.

3.3 Write Groups

Write groups reduce the JR seek cost without limiting the choices for the number of buckets. The buckets are divided across a number of write groups, each with an approximately equal number of buckets. A bucket segment belongs to a write group if its bucket belongs to the group. During each batch write, the bucket segments belonging to the same write group are placed in a contiguous area on disk. The batch write algorithm now proceeds as follows: (1)Allocate a contiguous area

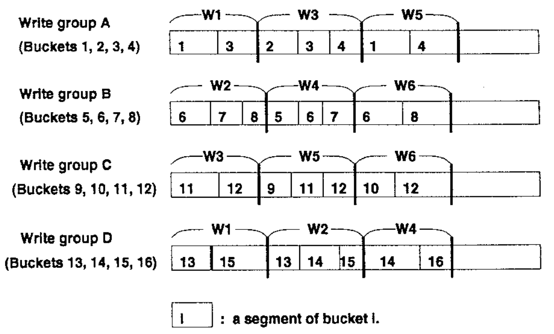

Figure 2: **A set of write groups. W1, W2, etc. denote successive batch writes.**

on disk for each write group. (2) When SB becomes full, choose some number of write groups to write to disk so that at least M_{wb} pages are reclaimed. Seek to the next empty disk block in the area designated for the write group. Choose some number of buckets in this group, and write all chosen buckets to the designated area in one sequential write. Since the number of disk seeks is the number of write groups written to disk in a batch write, we choose the largest write groups so that few groups need be written. Figure 2 show an example of write groups in a hash join.

Denote the number of write group as n_{wg}. Assuming the write group are approximately of equal sizes, the cost to read a write group is $A_{ran} + (\frac{|R|}{n_{wg}} - 1)A_{seq}$.

The cost to read a bucket is bounded by the cost to read its write group. $C(JR_R)$ is thus bounded by:

$$\begin{aligned} C(JR_R) \quad &\leq \quad \sum_{t=1}^{H_s} C(\text{read write group of } t) \\ &= \quad H_s(\frac{|R|}{n_{wg}} + \rho - 1)A_{seq} \end{aligned}$$

3.4 Read Groups

We can reduce read costs further with *read groups.* As in [5], we combine multiple buckets into one hash table during join phase. A set of buckets combined into one hash table is called a read group. Each read group is made as large as possible, but under two constraints. First, the size of a read group must be smaller than M_{SB}, so that the corresponding hash table can fit in SB. Second, the buckets in a read group must belong to the same write group. In other words, every read group is a subset of some write group. Bucket segment belongs to a read group if its bucket belongs to the read group.

The segments of a read group are read in the order they are written on disk, regardless of which buckets

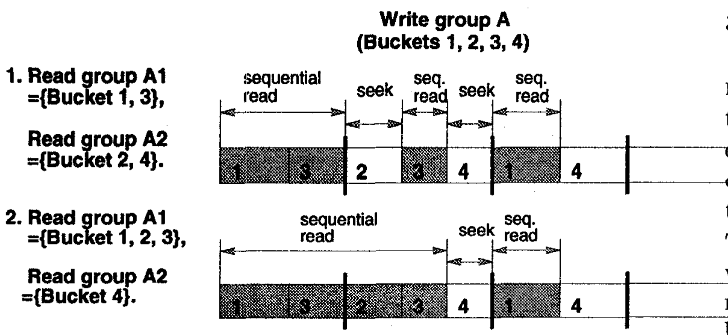
Write group A
(Buckets 1, 2, 3, 4)
1. Read group A1 ={Bucket 1, 3},
Read group A2 ={Bucket 2, 4}.
2. Read group A1 ={Bucket 1, 2, 3},
Read group A2 ={Bucket 4}.
sequential read
seek
seq. read
seek
seq. read
1
3
2
3
4
1
4
sequential read
seek
seq. read
1
3
2
3
4
1
4

Figure 3: **Two ways of assembling read groups from write group A and the operations to read the read group A1.**

they belong to. This way, the cost of reading all buckets of one read group is still bounded by the cost of sequentially reading its containing write group. Since many buckets can be assembled into one read group, the number of read groups is much smaller than the number of buckets, and the new reading cost much smaller.

Figure 3 shows the read groups that can be created for the write group A of Figure 2.

With n_{rg} read groups, $C(JR_R)$ becomes:

$$C(JR_R) \leq n_{rg}(\frac{|R|}{n_{wg}} + \rho - 1)A_{seq} \qquad (3)$$

The cost for reading outer relation S is again complicated by the smaller size of the input buffer. The number of additional disk seeks introduced because of the limited size of M_{IB} is bounded by[1] $\frac{|S|}{M_{IB}}$. The upper bound for the cost of reading relation S is thus:

$$C(JR_S) \leq n_{rg}(\frac{|S|}{n_{wg}} + \rho - 1)A_{seq} + \frac{|S|}{M_{IB}}(\rho - 1)A_{seq} \qquad (4)$$

If the S buckets were stored in a separate, dedicated disk, the disk head would remain at the location of its last read, and reading S buckets in chunks of M_{IB} would introduce no additional costs.

The cost of PW rises slightly when using write groups, because each write group participating in a batch write will introduce a disk seek. The average number of write groups participating in a batch write is bounded by $\max(1, n_{wg}\frac{M_{bw}}{M_{SB}})$. Let $H = \frac{M_{bw}}{max(1, n_{wg}\frac{M_{bw}}{M_{SB}})}$. $C(PW)$ should be modified to:

$$C(PW) \ = \ (|R| + |S|)\frac{H + \rho - 1}{H}A_{seq}. \qquad (5)$$

[1]We use a looser upper bound than derived in Section 3.2.2 to simply ensuing derivations.

3.5 Number Read and Write Groups

The number of read groups n_{rg} is known only dynamically during the join phase, and is determined by the source relations, the partition function and memory size. However, an upper bound for n_{wg} can be derived by classifying read groups into those smaller than $\frac{M_{SB}}{2}$, and those larger than or equal to $\frac{M_{SB}}{2}$. There is at most one read group smaller than $\frac{M_{SB}}{2}$ within each write group. If there are multiple such read groups in a write group, we can continue combining them into larger read groups, till either all or all but one read group are larger than $\frac{M_{SB}}{2}$. The number of read of groups larger than $\frac{M_{SB}}{2}$ is apparently smaller than $\frac{|R|}{M_{SB}/2}$. Therefore, the number of read groups is bounded by $n_{rg} \leq \frac{2|R|}{M_{SB}} + n_{wg}$.

Using this upper bound, and formulas 3, 4 and 5, we have a upper bound for the bucket I/O cost. The exact bucket I/O cost depends on the input data and is thus hard to minimize. However, we can minimize this upper bound to obtain a good performance. It can be shown that when $n_{wg} = \sqrt{\frac{2(|R|^2+|S|^2)}{(|R|+|S|+2M_{SB})\rho-1}}$, this upper bound for the bucket I/O cost is the minimum:

$$C(PW + JR) \leq$$
$$2(|R| + |S|)\frac{M_{SB}+\rho-1}{M_{SB}}A_{seq} + \frac{|S|}{M_{IB}}(\rho - 1)A_{seq} +$$
$$2\frac{\sqrt{2(\rho-1)(|R|+|S|+2M_{SB})(|R|^2+|S|^2)}}{M_{SB}}|R|A_{seq}$$

4 Performance Analysis

In this section, we study the performance of Seq$^+$, and compare it with current hash join methods. Both IB and OB are set to the smaller of $(\rho - 1)$ pages and 10% of the memory size.

Since HH and DHGH have the best performance among current methods, and have similar cost functions, we assume a hypothetical hash join whose cost function is the lower bound of these two methods. We denote this algorithm by G-HH. In particular, we assume the size of the anchor buckets for G-HH to be always $M - (|R|/M) + 1$ regardless of the number of buckets. This size is the upper bound of the actual anchor bucket size for both HH join and DHGH join. Thus our cost model is biased in favor of these methods. We also assume the same fraction of the outer relation is hashed into the anchor bucket(s) as the inner relation.

Based on our analysis (see Section 2), the cost function for G-HH is:

$$C(PR) \ = \ (|R| + |S|) \cdot A_{ran}$$
$$C(PW) \ = \ (|R| + |S| - (2M - \frac{|R| + |S|}{M} + 2)) \cdot A_{ran}$$

$$C(JR) = (|R| - (M - \frac{|R|}{M} + 1)) \cdot A_{seq} + (|S| - (M - \frac{|S|}{M} + 1)) \cdot A_{ran}$$

$$C(JW) = |RES| \cdot A_{ran}$$

For simplicity, we assume the input and result relations are all of the same size, i.e., $|R| = |S| = |RES|$, and reckon relation and buffer sizes in numbers of memory pages. Figure 4 shows the cost of the join methods under different $|R|/M$ ratios, with $\rho = 10$, and $M = 500$. Seq$^+$ runs approximately 3 times faster than G-HH. Figure 5 shows the performance gain of Seq$^+$ over G-HH with $M = 500$ and $\rho = 5$, 10 and 30, respectively. As expected, the improvements of Seq$^+$ over G-HHincrease with ρ, since Seq$^+$ incurs much less random I/O than G-HH. Even when ρ is as low as 5, Seq$^+$ still runs more than twice faster than G-HH.

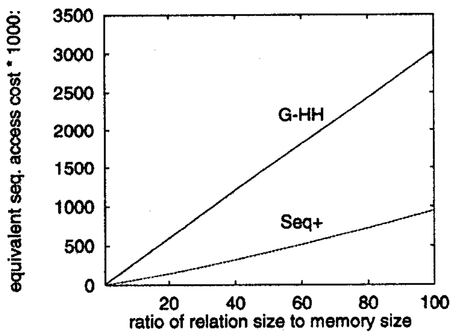

Figure 4: **Total cost of hash join methods.**

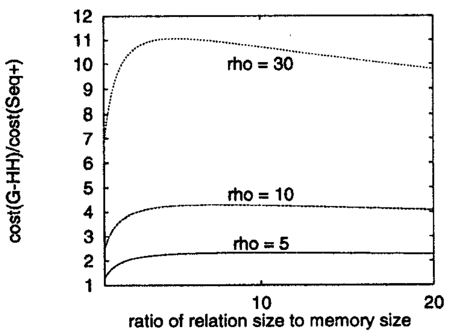

Figure 5: **Ratio of total costs of G-HH and** Seq$^+$**.**

Seq$^+$ allocates more space for IB and OB than does G-HH, thus leaving less memory for the staging buffer. To test the effects of memory consumption by IB and OB, we compare the algorithms under smaller memory ($M = 100$ and $\rho = 10$). As Figure 6 shows, in this case Seq$^+$ uses only 80% of memory as its staging buffer, while G-HH uses 98%. Seq$^+$ does not outperform G-HH by as much as in earlier cases since it uses a smaller SB. However, it still runs more than twice as fast as G-HH overall.

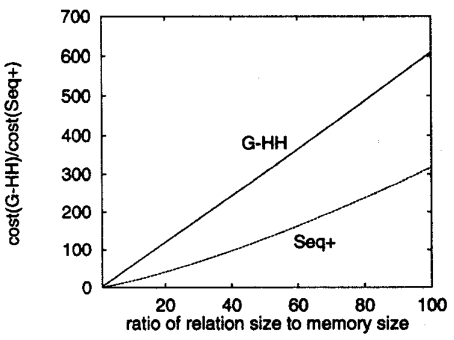

Figure 6: **Total cost of hash join methods, M = 100.**

The cost ratio of G-HH to Seq$^+$ drops as the relation size grows larger relative to the memory size. This is as expected. As the relations grows larger, bucket sizes increase and opportunities for combining buckets into read groups diminish. Figure 7 shows the cross-over points for the costs of G-HH and Seq$^+$. Even for $\rho = 5$, Seq$^+$ outperforms G-HH cost for relations as large as 1500 times memory size. These relation sizes are well beyond the relation size expected to be run on any reasonable system. For $\rho = 30$, Seq$^+$ runs 50% faster than G-HH even when the relation size is as large as 5000 time the memory size.

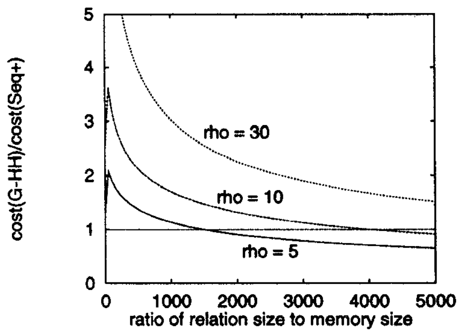

Figure 7: **Total costs of G-HH and** Seq$^+$**.**

We had assumed, based on the analysis in Section 2, that G-HH uses random I/O for PR, JW and reading relation R in the join phase. But what if we assume that a piece of data can somehow be read sequentially with a one-page input buffer as long as it is stored contiguously on disk, and that the result relation can somehow be written to disk with sequential block access with a one-page output buffer. This assumption further favors G-HH, but even under this assumption, Seq$^+$ still performs significantly better than G-HH. Figures 8 shows the ratio of G-HH and Seq$^+$ under the new assumption.

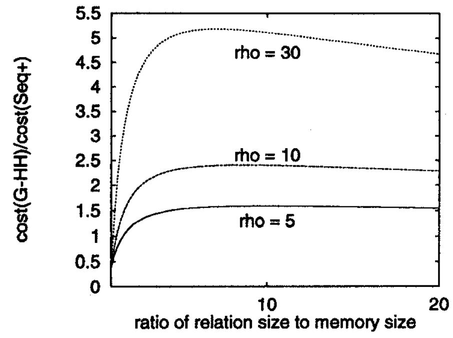

Figure 8: **Total cost ratios of G-HH and** Seq$^+$**, assuming sequential I/O in PR, JR and join-phase read of** S.

5 Conclusions

This paper proposes a new hash join method, Seq$^+$, that converts much of the random I/O to sequential I/O. This method increases input and output buffer sizes, and stores hash buckets on disk as segments, enabling the use of batch writes to reduce the write cost for buckets. It also introduces the idea of organizing buckets into read and write groups to reduce bucket read costs. In addition, it presents guidelines for choosing various algorithm parameters, and describes a cost model. Our method performs many times better than current hash join algorithms.

Since we have been interested in demonstrating the principles underlying our method, we have not explored various inherent optimization opportunities. For example, during the partition phase, we need no space for OB, and space can be shared between IB and SB. If we reserve H_s pages as bucket buffers, other SB pages can be used for IB. As tuples are consumed from IB, IB pages can switched to SB. Because of page sharing, the effective sizes of both IB and SB in the partition phase are larger than modeled by our cost function.

We have also used simple heuristics to determine the IB, OB and SB sizes. It is possible determine M_{IB}, M_{OB}, and n_{wg} simultaneously by minimizing the total cost function. Also, our method requires no specific choice of H_s for its performance, as long as $H_S \leq M_{SB}$. Since our method reserves 80% or more of memory for SB, the range of choice for H_s is large and the stability against bucket overflow is very close to that of DHGH join.

We emphasize that we have used an upper bound as the cost function for Seq$^+$ in our performance study. The actual performance gains of our method should be thus be even higher than claimed in this paper.

References

[1] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Computing*, vol. 1, no. 1, pp. 66–74, 1983.

[2] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, "Implementation techniques for main memory database systems," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 1–8, 1984.

[3] D. J. DeWitt and R. Gerber, "Multiprocessor hash-based join algorithms," in *Proceedings of VLDB 85*, pp. 151–164, Stockholm, 1985.

[4] M. Nakayama, M. Kitusregawa, and M. Takagi, "Hash-partitioned join method using dynamic destaging strategy," in *Proceedings of the 14th VLDB Conference*, pp. 468–478, 1988.

[5] M. Kitsuregawa, M. Nakayama, and M. Takagi, "The effect of bucket size tuning in the dynamic hybrid grace hash join method," in *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pp. 257–266, Amsterdam, 1989.

[6] L. D. Shapiro, "Join processing in database systems with large main memories," *ACM Transactions on Database Systems*, vol. 11, no. 3, pp. 239–264, September 1986.

[7] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Computing Surveys*, vol. 24, no. 1, pp. 64–113, March 1992.

[8] P. M. Chen, E. K. Lee, C. A. Gibson, R. H. Katz, and D. A. Patterson, "Raid: High-performance, reliable secondary storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–185, June 1994.

[9] J. Cheng, D. Haderle, R. Hedge, B. Iyer, T. Messinger, C. Mohan, and Y. Wang, "An efficient hybrid join algorithm: A DB2 prototype," in *Proceedings of International Conference on Data Engineering*, pp. 171–180, 1991.

[10] M.-L. Lo and C. V. Ravishankar, "Spatial joins using seeded trees," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 209–220, Minneapolis, MN, May 1994.

[11] D. L. Davison and G. Graefe, "Memory-contention responsive hash joins," in *Proceedings of the 20th VLDB Conference*, pp. 379–390, Santiago, Chile, 1994.