An Evaluation of Non-Equijoin Algorithms*

David J. DeWitt Jeffrey F. Naughton Donovan A. Schneider

Abstract

A non-equijoin of relations R and S is a *band join* if the join predicate requires values in the join attribute of R to fall within a specified band about the values in the join attribute of S. We propose a new algorithm, termed a *partitioned band join*, for evaluating band joins. We present a comparison between the partitioned band join algorithm and the classical sort-merge join algorithm (optimized for band joins) using both an analytical model and an implementation on top of the WiSS storage system. The results show that the partitioned band join algorithm outperforms sort-merge unless memory is scarce and the operands of the join are of equal size. We also describe a parallel implementation of the partitioned band join on the Gamma database machine, and present data from speedup and scaleup experiments demonstrating that the partitioned band join is efficiently parallelizable.

1 Introduction

In this paper we consider evaluation algorithms for a class of non-equijoins that we call "band joins." A band join between relations R and S on attributes $R.A$ and $S.B$ is a join in which the join condition can be written $R.A - c_1 \leq S.B \leq R.A + c_2$. The constants c_1 and c_2 may be equal, and one of the two may be zero. We use the term "band" because a tuple r in R joins with a tuple s in S only if $r.A$ appears within a "band" of size $c_1 + c_2$ about $s.B$. To the best of our knowledge, currently systems implement band joins using either nested loops or sort-merge. We propose a new algorithm, the partitioned band join algorithm, and present both analytic and experimental evidence (from an implementation on top of the WiSS storage system [CDKK85]) that it is faster than sort-merge over a wide range of band join queries and memory sizes. Another desirable property of the partitioned band join algorithm is that it maps well to shared-nothing parallel database machines. We also present experimental scaleup results from our implementation on the Gamma database machine [DGS+90].

Band joins arise in queries that require joins over continuous "real world" domains such as time or distance. For example, if the tuples in R and S represent events, and the attributes $R.A$ and $S.B$ represent the times at which events occur, then finding all pairs of events that occurred at nearly the same time will entail a band join. Note that unless the clocks measuring events in R and S are exactly synchronized, even finding events that occurred at the "same time" will require a band join, where the band is large enough to capture the skew between the two clocks.

From an algorithmic point of view, band joins are interesting because they present challenges not present in equijoins. There is a growing body of both analytic and experimental evidence that hash-based algorithms are highly effective for equijoins, surpassing the performance of sort-merge and nested-loops almost everywhere. Unfortunately, hash-based algorithms are ineffective for band joins, since the join condition involves ranges of values rather than exact matches of values. Furthermore, there is no way to use hashing to partition R into disjoint subsets $R_1, R_2, \ldots, R_k$, and S into disjoint subsets $S_1, S_2, \ldots, S_k$, such that the band join can be computed by the union of the joins $R_i \bowtie S_i$.

The algorithm we develop in this paper, the partitioned band join algorithm, works by partitioning R and S in such a way that the S partitions overlap both portions of other S partitions and portions of multiple R partitions. The basis for the partitioning lies in finding the quantiles for the join attribute of R; to avoid fully sorting R, we find these quantiles by sampling. We compute the number of samples required to find the quantiles to the required accuracy and confidence by using the Kolmogorov test statistic. The cost of this sampling is included in both our analytic and experimental results.

A critical parameter of a band-join is the number of tuples of R and S that fit within one band. (Two tuples t_1 and t_2 appear in the same band if the constants in the join attribute in each are within $c_1 + c_2$ of each other.) In this paper we assume that the bands are "small" in the sense that the number of tuples that fit in one band will also fit in memory. This is the most interesting case to consider; if the number of tuples that fit in one band will not fit in memory, then the join result will be huge, and gains due to clever evaluation algorithms will be swamped by the cost of writing the result relation to disk.

The rest of this paper is organized as follows. Section 2 describes the partitioned band join algorithm, while Section 3 describes our adaptation of the sort-merge join algorithm to band joins. Section 4 presents an analytic comparison between the partitioned band join algorithm and the classical sort-merge join algorithm applied to band joins. In Section 5 we describe an implementation of the algorithms and present results from experiments with the implementation in Section 6. Section 7 describes how the partitioned band join algorithm can be adapted for a shared-nothing multiprocessor and gives speedup and scaleup results from the implementation on Gamma.

2 Partitioned Band Join

The partitioned band join algorithm works by splitting up R and S into partitions R_i and S_i, then computing the band join by joining R_i and S_i for each i. The algorithm achieves its high performance by carefully choosing the partition sizes and overlaps, by performing the partitioning without sorting either R or S, and by using an efficient method for computing the subjoins between R_i and S_i.

2.1 Overview of the Partitioned Band Join Algorithm

A primary goal of the algorithm is to try to minimize the number of disk accesses by guaranteeing that pages never need to be re-read during the join of R_i with S_i. We can achieve this goal by ensuring the following two conditions:

1. Each of the R_i fits entirely into the buffer pool, and
2. For every tuple r in R_i, all tuples of S that join with r appear in S_i.

When these two conditions are satisfied, then we can join R_i and S_i by reading R_i into memory, then reading in S_i one page at a time, joining all tuples on each page of S_i with all of R_i before reading the next page of S_i.

We ensure the first condition by choosing partitioning elements by sampling R. This process is described more fully in Subsection 2.2. Here we consider the second condition, which determines the required overlaps between the partitions.

If condition two above is to be satisfied, then it must be the case that the range of tuples in S_i overlaps the range of tuples in R_i. The precise requirement is that if h_i is the greatest element appearing in $R.A$ in some tuple of R_i, and l_i is the least element appearing in $R.A$ in some tuple of R_i, then S_i must contain all tuples s such that $l_i - c_1 \leq s.B \leq h_i + c_2$ (A and B are the join attributes of R and S, respectively). Since it is possible that $h_i = l_{i+1}$, this implies that the range for S_{i+1} must overlap the range for S_i by $c_1 + c_2$.

Assuming that the partitioning values have been determined (by sampling), the tuples of the relations must actually be partitioned into the R_i and the S_i. There are two ways to do this, which we term "hybrid" and "Grace" partitioning, after the corresponding partitioning methods for hash-based equijoins [DKO+84, KTMo83]. Grace partitioning works by allocating a number of buffer pages equal to the number of partitions. Then each page of the relation is read into an input buffer, and each tuple t on the page is copied into the buffer page for the partition in which t belongs. To determine which partition a tuple t belongs, we binary search a table of partitioning values (constructed by sampling, as described below). The table contains the join attribute values that mark the boundaries between the R partitions. As a buffer page for a partition is filled up, it is written to disk. Note that when partitioning S the tuples in the overlapping portions of the S partitions are copied into multiple buffer pages, since these tuples must appear in multiple consecutive partitions.

Suppose that s is determined (by consulting the partitioning table) to fall in partition S_i, and suppose again that we are computing the join $S.B - c_1 \leq R.A \leq S.B + c_2$. Furthermore, let the value x_{i+1} divide partitions R_i and R_{i+1}. To see if s also belongs in S_{i+1}, we check if the join attribute of s, denoted $s.B$, satisfies $x_{i+1} - c_1 \leq s.B \leq x_{i+1} + c_2$. This can be seen by noting that the largest value in S_i is $x_{i+1} + c_2$, while the smallest value in S_{i+1} is $x_{i+1} - c_1$.

Hybrid partitioning works in the same way as Grace partitioning, except that enough buffer pages are allocated to keep all of R_1 in memory during the partitioning phase. Then after R has been partitioned, partition R_1 remains in memory, while partitions $R_2, \ldots, R_N$ are on disk. When S is being partitioned, if an S tuple s falls in partition S_1, it is immediately joined with R_1 rather than written to disk. The goal of hybrid partitioning is to avoid re-reading R_1 between the partitioning and joining phases.

Once the relations have been partitioned as described above, the band join problem is reduced to computing the individual joins $R_i \bowtie S_i$ for $1 \leq i \leq k$, where k is the total number of partitions. The basic idea for computing one of these subjoins, as mentioned above, is to read in R_i, then read in S_i one page at a time, joining each tuple of the current S page with all of R_i. To avoid scanning all of R_i for each S tuple, we first sort R_i using an in-memory sort. Then the join of each S tuple s with R_i can be accomplished by first binary searching R_i to find the first R tuple that joins with s, then scanning R_i until we pass the last R tuple that joins with s.

2.2 Sampling and Partitioning

To guarantee that each R_i fits in memory, we need to partition R into approximately equal sized partitions, each partition about the size of the buffer pool. The most straightforward way to partition R would be to sort R on $R.A$, then scan R to find the partitioning elements. This would incur the cost of a full sort of R during the partitioning phase, something that we wish to avoid. A better way is to randomly sample R to determine partitioning elements that with high probability are close to elements that would be found if we sorted R.

Suppose that we wish to partition R into k equal sized, disjoint partitions. We begin by taking n random samples (tuples) from R which are then sorted on $R.A$. Let the n sorted tuples be designated $r_1, r_2, \ldots, r_n$ in order of increasing $R.A$ value. If we wish to partition R into k partitions R_1 through R_k, we take $r_{n/k}.A$ to be the partitioning element between R_1 and R_2, $r_{2n/k}.A$ to be the partitioning element between R_2 and R_3, etc. By the Kolmogorov test statistic [Con71], with 99% certainty the percentile of each of the partitioning elements is off by at most plus or minus $1.628/\sqrt{n}$. For example, suppose we take 256 samples, recording the join attribute value of each sample, and then sort the resulting 256 values. If a value x appears at the 50% mark in the sorted list of samples, then with 99% certainty x appears between the 40th and 60th percentile in the sorted list of the join attribute of all tuples of R. Note that this error guarantee requires no assumptions about the distribution of the values in $R.A$; the Kolmogorov test is a non-parametric test that works equally well for any distribution.

Choosing the number of partitions is an interesting problem, perhaps best explained by an example. Suppose that R has $|R|$ pages, and that we have $|R|/3$ memory buffer pages available. Furthermore, suppose that we are using hybrid partitioning. The buffer pages required by hybrid during partitioning are of two types: those for partition R_1, and those for partitions $R_2, \ldots, R_k$. For the purposes of this example, we will ignore those pages for $R_2, \ldots, R_k$, since k will be small but the size of R_1 will not. (Our implementation does not ignore these pages, but including them complicates the exposition.)

In an ideal situation, we could choose $k = 3$, and R_1 would exactly fit in memory. Since our partitioning elements are only approximate, we cannot expect the partitions to be of equal size, so $k = 3$ is unreasonable. Note that if R_1 is actually larger than the buffer pool, the correctness of the algorithm is not affected; however, during both the partitioning and the joining phases performance will suffer due to buffer pool thrashing. Thus, for performance reasons, we need to pick k so that it is highly unlikely that R_1 will exceed the available buffer space.

The next logical choice is to set $k = 4$. In this case, the expected size of R_1 will be $|R|/4$ pages, so we are left with $|R|(1/3 - 1/4) = |R|/12$ buffer pages available to handle any overflow due to errors in the estimation of the partitioning elements. Now suppose we wish to be 99% certain that R_1 fits in the buffer pool. With n samples, the expected error in the quantiles is $1.628/\sqrt{n}$. Since a quantile is just a percentage, this means that the expected number of error pages is $1.628 * |R|/\sqrt{n}$. We need that this quantity is less than $|R|/12$, so the equation defining the number of samples required is

$$\frac{1.628 * |R|}{\sqrt{n}} \leq |R|/12$$

which implies that we must take at least $n = 382$ samples.

Still another choice would be to set $k = 5$. The same analysis as above shows that in this case we must take only $n = 150$ samples. However, now the expected size of R_1 is smaller by $|R|/4 - |R|/5 = |R|/20$. This in turn means that we can expect to do $|R|/20$ more reads and writes with $k = 5$ than with $k = 4$, since we save less by leaving a smaller R_1 in memory.

To summarize, there is a tradeoff between reducing non-sampling I/O by choosing k small, and reducing sampling I/O by choosing k large. To resolve this tradeoff, we have written an optimization procedure that takes as input $|R|$, the available memory size, the cost of a sample, and the cost of other I/O, and chooses a reasonable k and the number of samples required so that with 99% certainty there will be no thrashing of the buffers. We re-emphasize that this is not to say the join algorithm is 99% correct; it is always correct, the 99% merely refers to the probability that no paging of the buffer pool will be needed.

An interesting point to note is that the number of samples required does not depend upon $|R|$. Rather, it depends upon the ratio of $|R|$ to the available memory. This implies that if we scale $|R|$ and the available memory together (keeping the ratio constant) the cost of sampling relative to the cost of reading the relation diminishes.

3 Sort-Merge Band Join

The standard sort-merge join algorithm for equijoins can be adapted to handle band joins. However, in the case of band joins, the algorithm can be expected to "back up" much more often than in the case of equijoins, as it scans to pick up joining pairs of tuples. Our implementation of sort-merge was further complicated in that we used the optimization of skipping the final merge of each sort, performing the join on the final set of runs instead of on the two completely sorted relations. More detail on this optimization in particular, and on the band sort-merge algorithm in general, is given in Section 5. Here we focus on the question of how to handle backing-up in the final joining merge.

Suppose that we have sorted R and S down to their final set of runs (just before the merge that would produce the sorted relations). The general idea for the band merge-join is as follows: at all times, we have in the memory buffers 1) one page from each run of R, 2) one page from each run of S, and 3) a "window" of pages from the fully sorted S relation. Let the pages in this "window" of the sorted S relation be numbered $S[m]$, $S[m+1]$, ..., $S[m+k]$, where $S[m]$ is the most recent page of tuples merged out of the S runs.

At any given time, let r be the tuple with the smallest value in any of the R sorted runs. That is, if we were proceeding with the final merge in a sort of R, then r would be the next tuple to be added to the output. The tuple r is used to probe the tuples in the S window, searching for joining S tuples. If r does not join with any tuples in page $S[m+k]$, then $S[m+k]$ is eliminated from the buffer pool, since no subsequent R tuple could join with any tuple in $S[m+k]$. Note that it is not necessary to write $S[m+k]$ to disk, since it will not be referred to again. If r joins with the last tuple in sorted order in page $S[m]$, then another page of S tuples, $S[m-1]$, is merged out of the S runs. Finally, r is deleted from the buffer page for the R run from which it was taken.

If r was the last tuple on this page, the next page from that R run is read into memory.

4 Analytic Comparison

In this section we give simple cost formulas for evaluating the relative performance of sort-merge band join and both variants of the partitioned band join algorithm (that is, using hybrid and Grace partitioning.) The formulas below omit the cost of creating the answer tuples and writing the answer, since this cost will be similar for both algorithms.

For both algorithms, we used the following set of parameters:

COMP	0.001	ms. to compare keys
KEYSWAP	0.003	ms. to exchange two keys
MOVE	0.010	ms. to move a tuple
SWAP	0.030	ms. to swap two tuples
IOSEQ	10.0	ms. to do a sequential IO
IORAND	25.0	ms. to do a random IO

Furthermore, assume that there are B tuples per page, let R contain $|R|$ pages, and let S contain $|S|$ pages, and let F be the fraction of R pages that fit in memory.

For the sort-merge band join, assuming that the memory is large enough so that both relations can be sorted in two passes each, the I/O cost consists of three parts:

$(|R|+|S|)*IOSEQ$ to read the relations
$+(|R|+|S|)*IOSEQ$ to write the initial runs
$+(|R|+|S|)*IORAND$ to re-read initial runs

Assuming that when forming the initial runs we sort (join attribute, pointer) pairs using some $n \log n$ internal sort and then copy the runs into sorted order (in memory), the CPU cost for the algorithm is

$|R|*B*\log(|R|*B*F)*(KEYSWAP+COMP)$
// form initial R runs
$+|S|*B*\log(|S|*B*F)*(KEYSWAP+COMP)$
// form initial S runs
$+(|R|+|S|)*MOVE$
// copy to sorted positions
$+|R|*B*\log(|R|/F)*(COMP+SWAP)$
// merge R runs
$+|S|*B*\log(|S|/F)*(COMP+SWAP)$
// merge S runs

The total cost of the algorithm is the sum of the CPU and IO costs.

For the Grace partitioned algorithm, the I/O consists of four parts. Letting s be the number of samples taken, and k the number of partitions, the I/O cost is

$s*IORAND$
// initial sampling
$+(|R|+|S|)*IOSEQ$
//to read the relations
$+(|R|+|S|)*IORAND$
// to write the partitions
$+(|R|+|S|)*IOSEQ$
// to re-read partitions

The CPU cost is given by

$(|R|+|S|)*B*\log(k)*COMP$
// find partition
$+(|R|+|S|)*B*MOVE$
// copy to output partition
$+|R|*B*\log(R*B/k)*(KEYSWAP+COMP)$
// sort R partitions
$+|R|*B*MOVE$
// copy to sorted order
$+|S|*B*\log(R*B/k)*COMP$
// find first joining R tuple

Again, the total cost is the sum of I/O and CPU costs.

Finally, for hybrid partitioning, the cost is

$s*IORAND$
// initial sampling
$+(|R|+|S|)*IOSEQ$
// to read the relations
$+((k-1)/k)*|R|+|S|)*IORAND$
// to write the partitions
$+((k-1)/k)*|R|+|S|)*IOSEQ$
// to re-read partitions

The CPU cost is the same as the CPU cost for Grace partitioning, and again the total cost is the sum of I/O and CPU costs.

We tested these equations for a wide variety of parameters, and also tested similar cost formulas for two other algorithms, nested-loops and a variant of the partitioned band join algorithm in which for $1 \le i \le k$, both R_i and S_i are sorted and are simultaneously memory resident. Since these two algorithms were worse than sort-merge and the basic partitioned band join algorithms everywhere, we did not pursue them further. A representative graph of these cost formulas for various fractions of R fitting in memory appears in Figure 1. In that graph, both R and S had 500 pages, and 40 tuples per page. The optimal number of samples and partitions for each memory configuration were computed by the same optimization procedure used in our implementation of the partition band join algorithm. Since the model gave performance results similar to our implementation, we defer a discussion of the performance results until Section 6.

5 Implementation Details

In order to evaluate the relative performance of the sort-merge and partitioned band join algorithms, each was implemented using the single user version of WiSS [CDKK85]. The services provided by WiSS include sequential files, byte-stream files as in UNIX, B^+ tree indices, long data items, an external sort utility, and a scan mechanism. A sequential file is a sequence of records that may vary in length (up to one page) and that may be inserted and deleted at arbitrary locations within a file. Optionally, each file may have one or more associated indices that map key values to the record identifiers of the records in the file that contain a matching value. One indexed attribute may be designated to be a clustering attribute for the file.

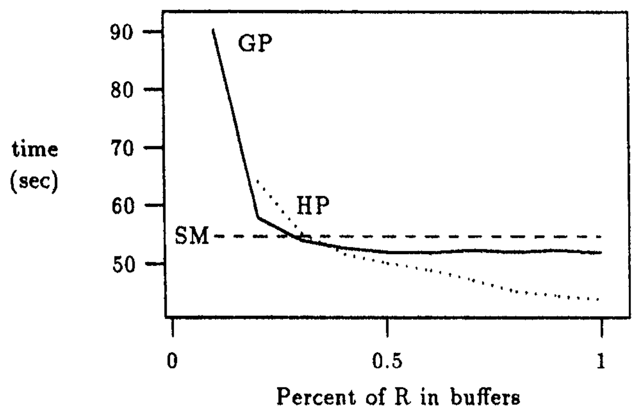

Figure 1: Model comparison between GP, HP, and SM.

Rather than using the standard WiSS buffer pool (which uses an LRU replacement policy) to buffer data pages, each of the join algorithms explicitly managed its own buffer space. Several factors motivated this decision. First, doing so simplified the task of varying the amount of buffer space available for a query without having to recompile WiSS each time. Second, since the WiSS sort code (which we intended to use as the basis for the sort-merge band join algorithm) already managed its own buffer space, doing the same thing for two partitioning algorithms seemed the fairest thing to do. Finally, allowing each algorithm to carefully manage replacement of pages directly, ensures that each algorithm is evaluated in the best possible light.

In order to avoid the difficulties of gathering reproducible results on a time-shared system with multiple users, file system buffering, and virtual memory paging, we elected instead to use a single node of the iPSC-2 hypercube on which the Gamma database system is implemented. Each node has a 386 processor, 8 megabytes of memory, and a 330 megabyte disk; more details are given in Section 7.

5.1 Hybrid and Grace Partitioned Band Join Algorithms

As described in Section 2, both the hybrid and Grace band join algorithms begin by splitting the the two relations to be joined, R and S, into partitions R_i and S_i, for $1 \leq i \leq N$. The inputs to the partitioning operator include N, the number of partitions, a description of the partitioning (join) attribute (type, length, offset), an N-element partitioning vector that specifies the upper and lower bounds of each partition (as discussed in Section 2, these bounds are produced by sampling the inner relation, R), and the number of buffer pages to be used to hold tuples of each partition during splitting process. With the hybrid algorithm, typically one page is allocated to partitions 2 to N with the remaining buffer space being used for partition 1. In the case of the Grace algorithm, each partition is allocated the same number of buffers, typically 1 or 2, depending on the number of partitions selected during the sampling phase.

Sampling was implemented by randomly generating a key value for the relation, and retrieving the tuple with that value. For implementational convenience, we used a dense index on the key attribute for this purpose, although the ability to take random samples from a relation does not depend upon this assumption [OR89, ORX90].

Our sampling technique effectively means at least one I/O per sample (more if the pages forming the upper levels of the index are not resident in the buffer pool.) Optionally, we could sample at the page level, using all tuples on a page when it is brought in. (Page level sampling has been proposed in [HOT88] for the purpose of join and selection selectivity estimation.) If the tuples on each page are not correlated on their join attribute, page level sampling is very effective, and would reduce the sampling overhead in our algorithms by a factor equal to the number of tuples per page. We did not implement this alternative, so our performance figures are "worst-case" numbers for sampling.

Partitioning the relations proceeds as described in Section 2. However, as an important optimization, as the inner relation (R) is being partitioned, a "range-vector" filter is formed containing the actual minimum and maximum attribute values of each partition. As each tuple is added to its partition, its join attribute value is compared to the current minimum and maximum values for the partition to determine if the value constitutes a new minimum or maximum. This is most significant for the "first" and "last" partitions — for example, the range-vector only gives an upper bound on the join attribute values for tuples in the first partition. If the actual smallest value appearing in the join attribute of the first partition is x, and the band is c, then any tuple in the outer (S) relation with join attribute less than $x - c$ can be discarded, since it could not possibly join with any tuple of the inner relation. This filter is employed while partitioning the outer relation much as bit vector filters are used when processing equijoins [DG85, SD89].

Once both relations have been partitioned, the actual join proceeds as follows (the differences between the Grace and hybrid algorithms will be discussed below). For each partition, the pages of the corresponding partition of the inner relation are read into memory and sorted (actually, instead of sorting entire tuples, pointers to the tuples are sorted). Next, the pages of the corresponding outer partition are processed. For each outer tuple, a lower (upper) bound is computed by subtracting (adding) the band value from (to) the tuple's join attribute value. The lower bound is then used to perform a binary search of the sorted inner partition to determine the appropriate starting tuple. Beginning with this tuple, the outer tuple is then joined with all subsequent inner tuples whose join attribute values are greater than the computed upper bound. As result tuples are produced they are blocked into pages and written to the result relation.

With the Grace algorithm all partitions are treated identically. In the case of the hybrid algorithm, the partitioning of the inner relation sorts partition 1 and leaves it resident in memory rather than writing it back to disk. Then, as the outer relation is partitioned, tuples that overlap the range of partition 1 are joined immediately rather than being written to disk. Partitions 2 to N are processed in the same way as they are with the Grace algorithm.

5.2 Sort Merge Band Join Algorithm

The sort-merge equijoin algorithm begins by sorting both relations on the join attribute. Then, the two sorted relations are scanned, joining tuples with equal join attribute values. While each tuple of the "outer" relation is examined only once, if the join attribute values of the outer relation are not unique, the scan of the inner relation must be "backed up" in order to produce the correct result. As mentioned in Section 3, adapting this algorithm to handle band joins is straightforward except that since each outer tuple joins with a band of tuples from the inner relation, the scan of the inner relation needs to be backed up after almost every single outer tuple.

In the past our comparisons of the relative performance of the sort-merge and hybrid equijoin algorithms have sometimes been criticized [Gra] for not being totally fair. In particular (as we ourselves first observed in [DG85]), it is possible to improve the performance of the sort-merge join algorithm (for both equi- and band-joins) by combining the final merge phase of both sort steps with the actual join phase, avoiding reading and writing the final runs of the two sorted relations. For this paper, we implemented this modified merge-join algorithm as discussed below.

Our sort-merge band join algorithm operates as follows. As with the hybrid and Grace algorithms, the algorithm manages its own buffer space so that the replacement of pages can be directly controlled. The algorithm begins by performing a partial external merge sort of the inner relation. Given a $K+1$ page buffer, the initial runs of K pages are sorted in memory using the same pointer-based, quick-sort algorithm used to sort the inner partitions of the two partitioning algorithms. Runs are then merged using a K-way merge until the first two pages of the final sorted run are produced. (This implicitly assumes that if there are B tuples per page, then there are no more than B tuples in the band. This was true for all our test cases.) At this point the sort of the inner relation is "suspended" until more inner tuples are needed (this will become clearer below).

Next, the sort of the outer relation is initiated. This sort is processed in a similar fashion to the sort of the inner relation except that as the final sorted run is being formed outer tuples are immediately joined with the appropriate tuples of the inner relation. Additional tuples from the inner relation are produced "as needed" during this join process by reactivating the sort of the inner relation to produce the next page of sorted inner tuples. (In effect, the sort of the inner relation and the sort/join of the outer relation are implemented as co-routines.) Pages of the sorted inner relation are discarded as soon as it can be safely determined that the tuples they contain cannot possibly join with any additional tuples from the outer relation. Result tuples are blocked into pages and written to the output relation.

It is important to understand that this performance "optimization" does not come totally for free. In particular, during the join phase, one must allocate input buffers for runs of both the inner and outer relations as well as several buffers for the merged input tuples and the output tuples.

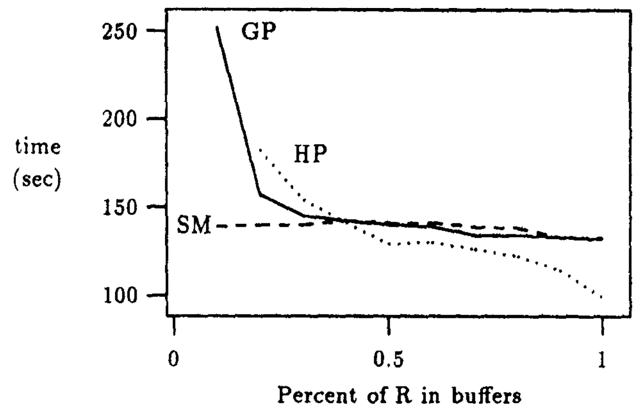

Figure 2: 20K tuples join 20K tuples, answer 20K tuples.

6 Uniprocessor Experiments

In this section we describe a series of experiments that we ran on our uniprocessor implementation of the band join algorithms. In all cases the test relations were based upon the Wisconsin Benchmark relations [BDT83], with some fields modified to be more useful in testing band joins. The new fields will be discussed as they are used in the experiments below. The tuple size was 188 bytes. Furthermore, in each of the experiments below an answer tuple t was defined to be the concatenation of the pair of tuples that joined to produce t, so answer tuples were 376 bytes long. In all cases we used a symmetric band, that is, $c_1 = c_2$.

For the first experiment we ran, we used the two attributes **hundreds** and **hundredsPlus1**. In a relation with N tuples, the attribute **hundreds** contains the numbers 0, 100, ..., $100*(N-1)$ in random order, while **hundredsPlus1** contains the numbers 1, 101, ..., $100*(N-1)+1$, again in random order. In this experiment, each relation had 20,000 tuples; we joined column **hundreds** in R with **hundredsPlus1** in S with a band of size two ($c = 1$). This means that each tuple of R joins with one tuple of S, and vice-versa. Figure 2 gives the results for memory sizes ranging from 1/10 of R in memory to all of R in memory. The curve for sort-merge is labeled SM; the curves for Grace partitioned band join and the hybrid partitioned band join are labeled GP and HP, respectively.

Note the similarity in the shapes and relative positions of the curves to those generated by the analytical model, shown in Figure 1. The absolute values for the plots in the graphs differ for a number of reasons, primarily because 1) the graph in Figure 1 omits the cost of forming and writing the answer, and 2) we made no attempt to do an exact match of the hardware parameters in the model with the hardware parameters of our implementation.

The high cost for the Grace and hybrid partitioned band join algorithms when less than about 1/3 of R fits in the buffer pool is due to sampling overhead. Table 1 gives the percentage of execution time due to sampling for the Grace partitioned band join curve in Figure 2. Note that if we implemented page-level sampling (as described in Section 5) this overhead would be reduced by a factor of 43, the number of tuples per page.

mem size	sampling time	percent of total
0.1	103 sec.	40
0.2	20 sec.	13
0.3	7 sec.	5
0.5	3 sec.	2
1.0	0 sec.	0

Table 1: Sampling costs as a percentage of running time.

The next experiment we ran was designed to test the performance of the three algorithms when the input relation sizes differ. For this purpose, we used an R relation of 10K tuples, and an S relation of 100K tuples. For the join, we used the attribute `twenties` in R, and the attribute `twentyWrap` in S. These attributes are defined such that for these relation sizes, the R tuples contain the join attribute values 0, 20, ..., $20 * (10000 - 1)$ in random order, while S contains the join attribute values 0,1,..., 9, 20,21,...,29, ..., $20*(10000-1)$, $20*(10000-1)+1$, ..., $20*(10000-1)+9$. With a band of size 2 ($c = 1$), every R tuple joins with two S tuples, while 2/10 of S tuples join with one R tuple and 8/10 of S tuples join with no R tuple, giving a result size of 20K. A graph of the results of this experiment is presented in Figure 3. Note that although not all S tuples join with an R tuple, the ranges for the join attributes in R and in S are essentially the same, so range-filtering had no effect.

This graph illustrates several important properties of the algorithms. Most obvious is the benefit that both the partitioned band join algorithms gain from not having to sort a 100K relation (S). This is most apparent for the small memory data points. At memory equal to 0.1 of R pages, sort-merge had to make multiple passes over R and S before even beginning the final merge; it wasn't until memory equal to 0.5 of R that the large S relation could be sorted in two passes (with the join computed on the second pass.) It is also clear that hybrid performs much better than Grace for large memory sizes. There are two reasons for this: first, hybrid doesn't have to re-read the portion of R that falls in R_1. However, this is not very significant, since the cost of re-reading part of the 10,000 R tuples is dwarfed by the cost of reading the 100,000 S tuples and writing 20,000 answer tuples that are twice as large as the R tuples. Much more importantly in this case, the portion of S that falls in S_1 is never written to disk or read back in, because those tuples in S_1 are immediately joined with R_1 in the partitioning phase of the algorithm.

The final experiment we ran was designed to demonstrate the effect of range filtering, as described in Section 5. We again joined 20K tuples with 20K tuples, but this time the join attribute in R contained the values 0, 20, ..., $20 * (20000 - 1)$ (in random order), while the join attribute in S contained the values 0, 100, ..., $100*(20000-1)$ (again in random order). With a band of size 100 ($c = 50$), 1/5 of the S tuples join with 5 R tuples, and the remaining S tuples join with no R tuples, so the answer size is again 20K tuples. However, in this case the range for the join attribute

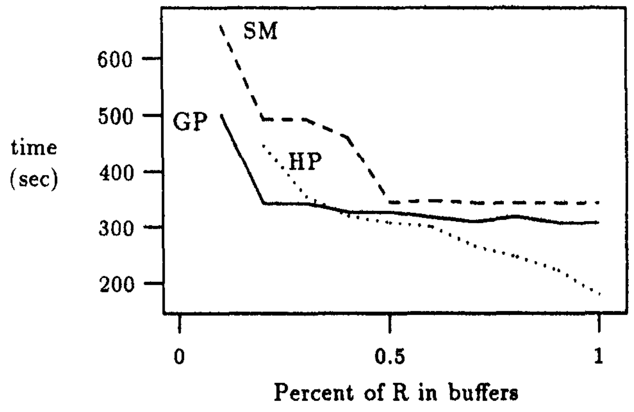

Figure 3: 10K by 100K, band 2, answer 20K tuples.

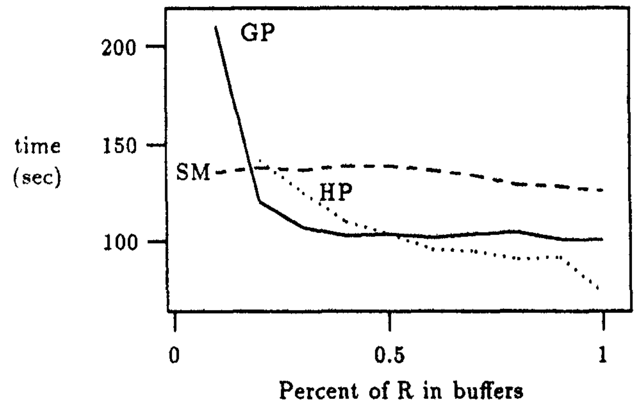

Figure 4: 20K tuples join 20K tuples, answer 20K tuples.

in the R relation is 1/5 the range of the join attribute for the S relation, so range filtering has a significant affect.

The result graph for this experiment appears in Figure 4. Both Grace and hybrid are able to "filter" 4/5 of the S tuples, which means that these tuples are just read once and thrown away. Hybrid does even better than Grace for large memory sizes, again since it 1) doesn't re-read R_1, and 2) joins S_1 without writing it or re-reading it. Hybrid is especially successful here, since for large memory sizes, all S tuples either fall in S_1 or are filtered out, so the result is that the join is computed by reading the pages of the S relation and never writing them.

7 Multiprocessor Experiments

In this section we consider the execution of band joins in a parallel environment. In order to simplify the implementation effort, we implemented only a parallel version of the hybrid partitioning band join algorithm.

7.1 Parallel Hybrid Partitioning Band Join

The Gamma Database Machine [DGS+90] served as our experimental vehicle. Gamma falls into the class of *shared-nothing* [Sto86] architectures. The hardware consists of a 32 processor Intel iPSC/2 hypercube. Each processor is configured with a 80386 CPU, 8 megabytes of memory, and a 330 megabyte MAXTOR 4380 (5 1/4 in.) disk drive. Each disk drive has an embedded SCSI controller which provides a 45 Kbyte RAM buffer that acts as a disk cache on sequential read operations.

The nodes in the hypercube are interconnected to form a hypercube using custom VLSI routing modules. Each module supports eight full-duplex, serial, reliable communication channels operating at 2.8 megabytes/sec. A custom operating system, NOSE, tailored especially for database processing, runs on each processor.

In Gamma, relations are *horizontally partitioned* [RE78] (also known as declustering [LKB87]) across all disk drives in order to increase the aggregate I/O bandwidth provided by the hardware. The query language of Gamma provides the user with several alternative declustering methods. For the experiments described below, the user determined which tuples reside on each site based on a range predicate applied to the partitioning attribute of each tuple of the relation. A collection of tuples stored on a processor is referred to as a *fragment* of the relation.

Extending the sequential version of the hybrid partitioning band join algorithm to a parallel environment was relatively straightforward. To simplify the implementation, each partition was mapped to an individual processor. In addition, we assumed that each partition of the inner relation was small enough to fit entirely in a processor's buffer pool.

The parallel version of the hybrid partitioned band join algorithm is as follows. First, each processor randomly samples its local fragment of the inner relation and sends the join attribute values of the sampled tuples to a central coordinator. The coordinator sorts all the sampled values and determines the partitioning elements such that the inner relation will be divided into as many buckets as there are processors. The coordinator then sends a copy of these partitioning elements to each processor whose disk contains a fragment of the inner relation. Each processor reads its local fragment of the inner relation and re-distributes it over the network using the partitioning elements. As tuples from the inner relation arrive at a processor, they are stored in memory and subsequently sorted. After this phase is complete, the outer joining relation is similarly re-distributed over the network using the partitioning elements derived from the inner joining relation. Of course, tuples that fall into a neighboring bucket due to the width of the band are replicated and sent to the processor that is handling this bucket. Hence, in a parallel environment, an increase in the size of the band results in increased network traffic. As tuples from the outer relation arrive at a processor, they are used to binary search the sorted inner tuples and compute any output tuples, exactly as was done in the uni-processor version of the algorithm.

One problem to overcome with this parallel algorithm is how to to correctly and efficiently sample the inner relation in parallel in order to determine the partitioning elements. For correctness, the relation must be sampled randomly as if it were stored on a single processor. That is, each tuple, regardless of the processor that it is stored on, must be equally likely to be sampled. Note that if we wish to take N samples, it is not acceptable to have each processor take $1/N$ samples, since this will not result in a truly random sample. To see this, note that if each processor takes $1/N$ samples, then we will never get a set of N samples in which more than $1/N$ tuples come from any single processor's portion of the database.

To take a truly random sample while still making use of the parallelism available, in our implementation, each processor attempts to sample N tuples from its local fragment of the relation (each processor uses the same random number generator with the same seed). However, for efficiency, each processor checks the local catalog information to determine if the tuple to sample is indeed stored on its local disk. If so, the tuple is retrieved from disk and its join attribute value is sampled. If the tuple is not stored locally, the sample can be ignored. In terms of disk I/O, the effect is the same as if some central processor generated N random keys, then sent to each processor p only the keys that for tuples in the partition stored at p. As in the uni-processor experiments, a B-tree index is used to efficiently retrieve the tuple to sample.

Note that this optimization does not require that the join attribute of the inner relation be identical to the attribute used to partition the inner relation during relation creation. Instead, it only requires that the attribute used to fetch a random tuple is the same attribute used to partition the inner relation during relation creation. Furthermore, if no such catalog information is available the algorithm still works correctly, it will only suffer a performance degradation due to unsuccessful searches of the index for tuples stored on other processors.

7.2 Experiments and Results

Scaleup and speedup are useful metrics for evaluating multiprocessor database machines [DG90]. Scaleup is an interesting metric for multiprocessor database machines as it indicates whether a constant response time can be maintained as the workload is increased by adding a proportional number of processors and disks. Speedup is an interesting metric because it indicates whether additional processors and disks result in a corresponding decrease in the response time of a query. A similar set of experiments were reported in [EGKS89] for equi-join queries on Release 2 of Tandem's NonStop SQL system and in [DGS+90] for equi-join queries in Gamma.

7.2.1 Scaleup

For the scaleup experiments, we varied the number of processors with disk from 1 to 30. At 1 processor, a 10,000 tuple relation was joined with a 100,000 tuple relation. At 10 processors, the relations were scaled to 100,000 tuples and 1,000,000 tuples, respectively. Similarly, at 30 processors, the sizes of

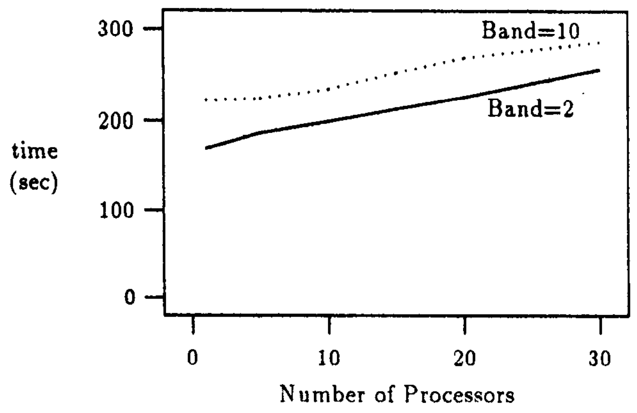

Figure 5: Scaleup performance.

the relations to be joined were 300,000 and 3,000,000 tuples. For every configuration, each of the relations to be joined was evenly distributed during relation creation amongst all the processors by applying a range predicate to the unique1 attribute (whose values range from 0 to the relation cardinality minus 1). The join query tested was the **twenties** join **twentyWrap**, as described in the uniprocessor experiment in Section 6.

Figure 5 presents the scaleup results for the parallel hybrid partitioning band join algorithm for band sizes of 2 and 10. Three factors contribute to the slight increase in response times. First, the task of initiating five processes at each site (two relation scans, a join, a store, and a sampling operator) is performed by a single processor. Second, as the number of processors increase, the effects of short-circuiting [DGS+90] messages during the execution of the query diminishes. For example, in the 5 processor configuration, approximately 1/5th of the tuples of the input relations and the result relation will be sent to a process on the same processor, thereby short-circuiting the communications network. As the number of processors is increased, the number of these short-circuited packets decreases to the point where, with 30 processors, only 1/30th of the packets will be short-circuited. Because these intra-node packets are less expensive than their corresponding inter-node packets, smaller configurations will benefit more from short-circuiting. Finally, the more processors added, the larger the skew in the sizes of the subjoins allocated to each processor.

This demonstrates another interesting tradeoff between sampling time and execution time. Roughly speaking, the more sampling, the lower the skew, hence the faster the execution time exclusive of sampling; but clearly, the more sampling, the higher the overhead of sampling. In more detail, as we increase the workload, the total relation size increases. Since the errors in the sizes of the partitions are proportional to the size of the total relation, this means that if the total number of samples is kept constant, then the expected error in the partitions will increase. In our implementation, we scaled the number of samples along with the number of processors, keeping the expected number of samples per processor constant. For example, the five processor join took five times as many samples as the one processor join. However, since the error is inversely proportional to square root of the number of samples, scaling the number of samples linearly is not sufficient to keep the expected error in the partitions constant. That is why we saw larger skews at the larger configurations.

To keep the skew constant as we add processors, we would have to scale the number of samples quadratically in the number of processors. Since the number of processors grows linearly, this means that the expected number of samples taken by each processor will also grow linearly. For example, in the 30 node configuration, if we wish the skew to be the same as in the two processor case, each processor will need to take, on average, 15 times as many samples as in the two node case. This implies that while we saw good scaleup to 30 processors, the parallel hybrid partitioning band join algorithm will not scale indefinitely.

7.2.2 Speedup

For the speedup experiments, we kept the size of the relations to be joined constant at 1,000,000 and 100,000 tuples while varying the number of processors from 5 to 30. We also held constant the number of samples used to determine the partitioning elements. We again used the **twenties** join **twentyWrap** query with band size 2 as our test query.

no. of processors	execution time	speedup
5	349.2	1.00
10	177.5	1.97
15	125.6	2.78
20	100.1	3.49
30	75.9	4.60

Table 2: Speedup results.

The response time and speedup for a band join of size two (result relation size of 200,000 tuples) are shown in Table 2. It is obvious that adding additional processors significantly reduces the execution time of the query. Several factors prevent the system from achieving perfectly linear speedups. (It is important to note that since the base case was 5 processors a perfect speedup factor for 30 processors would be 6.0 and not 30.0!) As was the case in the scaleup experiments, performance is limited by the overhead of scheduling the operators of the query tree, the effects of short-circuiting, and the effects of skew in the size of the subjoins allocated to each processor.

To demonstrate the effect of skew, we measured the number of tuples produced at each join site. We then took the maximum of these values and measured how far it differed from the optimal value (assuming a perfectly uniform distribution). In the 5 processor configuration, the maximum skew was approximately 5%. In the 30 processor configuration, though, the maximum skew was found to be 18% above optimal. Since in a multiprocessor, performance is limited by the slowest site, the increase in skew as processors are added results in sublinear speedups.

8 Conclusions

The two variants of the partitioned band join algorithm, Grace and hybrid, compare favorably to the optimized sort-merge band join algorithm. This is encouraging and perhaps somewhat surprising: while it has previously been demonstrated that the equijoin hash-based join algorithms outperform sort-merge, it was not initially obvious that these new algorithms would be as effective for band joins as the hash based algorithms are for equijoins. This is because, when compared to the hash-based equijoin algorithms, the partitioned band join algorithms do significantly more work: where the equijoin algorithm hashes a bucket of the inner relation, the partitioned band join algorithm sorts a partition of the inner relation; where the equijoin algorithm does a hash-based lookup, the partitioned band join algorithms do a binary search; and finally, the equijoin algorithms have no equivalent to the sampling overhead in the partitioned band join algorithms.

Unlike the situation for the hybrid and Grace hashed equijoin algorithms, Grace partitioned band join does not always dominate sort-merge, and hybrid partition band join does not always dominate Grace partitioned band join. The reason for this is the added cost of sampling; when memory is scarce, both the Grace and the hybrid variants of the partitioned band join algorithm must take a lot of samples to ensure that the errors in the partition sizes do not cause thrashing of the buffer pool. For small memory sizes, hybrid must sample more than Grace because there is effectively less memory available for R_1 in hybrid partitioning than there is for the partitions in the Grace algorithm. This implies that a system should probably have all three algorithms available for performing band joins; the optimizer must decide which algorithm is appropriate for a given band join.

The partitioned band join algorithms perform especially well in two cases: when a significant fraction (say, more than 50%) of one of the operands fits in memory, and when the input relation sizes are different. The latter is an especially important case, since it will often occur when the band join is part of a query of the form $\sigma(R) \bowtie S$. Finally, we have demonstrated that the partitioned band join algorithm is effective in multiprocessor systems, achieving good speedy and scaleup for configurations of at least 30 processors (the maximum we could measure.)

Acknowledgements

This research was funded by an IBM Research Initiation Grant. We would also like to thank Rick Rasmussen for his help both in porting the uniprocessor version of the three algorithms to the hypercube and in locating a particularly nasty Gamma bug.

References

[BDT83] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proc. of the Ninth VLDB Conf.*, pages 8–19, 1983.

[CDKK85] H-T. Chou, D. J. Dewitt, R. H. Katz, and A. C. Klug. Design and implementation of the Wisconsin Storage System. *Software—Practice and Experience*, 15(10):943–962, October 1985.

[Con71] W. J. Conover. *Practical Nonparametric Statistics.* John Wiley & Sons, New York, NY, 1971.

[DG85] D. M. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. of the Twelfth VLDB Conf.*, pages 151–164, 1985.

[DG90] D. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad. *SIGMOD Record*, 19(4), 1990.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. Scneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Trans. on Knowledge and Data Engineering*, 2(1), 1990.

[DKO+84] 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 *Proc. of the 1984 SIGMOD Conf.*

[EGKS89] S. Englert, J. Gray, T. Kocher, and P. Shah. A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large database. Technical Report 89.4, Tandem Part No. 27469, Tandem Computers, 1989.

[Gra] Goetz Graefe. Personal Commuication.

[HOT88] W. C. Hou, G. Ozsoyoglu, and B. K. Taneja. Statistical estimators for relational algebra expressions. In *Proc. of the 1988 PODS Conf.*

[KTMo83] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of hash to data base machine and its architecture. *New Gen. Comp.*, 1(1), 1983.

[LKB87] M. Livny, S. Khoshafian, and H. Boral. Multidisk management algorithms. In *Proc. 1987 SIGMETRICS Conf.*, 1987.

[OR89] F. Olken and D. Rotem. Random sampling from B+-trees. In *Proc. of the Fifteenth VLDB*, 1989.

[ORX90] F. Olken, D. Rotem, and P. Xu. Random sampling from hash files. In *Proc. ACM SIGMOD Conf.*, pages 375–386, 1990.

[RE78] D. Ries and R. Epstein. Evaluation of distribution criteria for distributed database systems. Technical Report UCB/ERL Tech. Rep. M78/22, UC-Berkeley, May 1978.

[SD89] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. of the 1989 SIGMOD Conf.*

[Sto86] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.