Practical Skew Handling in Parallel Joins

David J. DeWitt* Jeffrey F. Naughton* Donovan A. Schneider† S. Seshadri*

July 14, 1992

Abstract

We present an approach to dealing with skew in parallel joins in database systems. Our approach is easily implementable within current parallel DBMS, and performs well on skewed data without degrading the performance of the system on non-skewed data. The main idea is to use multiple algorithms, each specialized for a different degree of skew, and to use a small sample of the relations being joined to determine which algorithm is appropriate. We developed, implemented, and experimented with four new skew-handling parallel join algorithms; one, which we call *virtual processor range partitioning*, was the clear winner in high skew cases, while traditional hybrid hash join was the clear winner in lower skew or no skew cases. We present experimental results from an implementation of all four algorithms on the Gamma parallel database machine. To our knowledge, these are the first reported skew-handling numbers from an actual implementation.

1 Introduction

Multiprocessor database system technology has progressed to the point where a number of companies are now shipping products that use parallelism to provide dramatic speedup and scaleup performance. It is clear from the success of these systems that parallelism is an effective means of meeting the performance requirements of large database applications. However, the basic technique that these systems use for exploiting intra-query parallelism (hash-based redistribution of relations on their joining attribute) [DG92] is vulnerable to the presence of skew in the underlying data. Simply put, if the underlying data is sufficiently skewed, load imbalances in the resulting parallel join execution will swamp any of the gains due to parallelism and unacceptable performance will result.

In response to this problem, a large and growing number of skew-handling algorithms have been proposed. In general terms, these algorithms do a significant amount of preprocessing in order to compute an execution plan designed to minimize load imbalances. While these algorithms may succeed in minimizing skew, invariably they perform much worse than the basic parallel hash join algorithm on data that is not skewed. For example, most of the previously proposed skew handling algorithms require that the relations to be joined are completely

*Department of Computer Sciences, University of Wisconsin-Madison.

†HP-Labs, Palo Alto.

scanned before the join begins [HL91, WDYT90, KO90]. Since the time to perform a parallel hash join is a small multiple of the time required to scan the two relations being joined, this can represent a substantial overhead, which is unacceptable for anything but extremely skewed data.

Since there little or no empirical evidence that extreme degrees of skew occur commonly in practice, it is sub-optimal to penalize the normal case in order to benefit an extreme case. For this reason, we sought to develop an approach to join processing in which the "normal" case approaches the performance of the fastest known parallel join algorithms on non-skewed data, but that avoids the disastrous performance degradation that standard hash-based join processing suffers on skewed data.

The basic idea in our approach is that we have multiple algorithms, each optimized for differing degrees of skew. We found in our experiments that two algorithms are sufficient: the usual parallel hybrid hash join algorithm [SD89], and a new algorithm that we call *virtual processor range partitioning*, performs well on moderately skewed data at a cost slightly higher than that of the parallel hybrid hash join. Before settling on these two algorithms, we implemented three other new skew handling algorithms (range partitioning, weighted range partitioning, and a scheduling version of virtual processor range partitioning) and performed tests on the implementation. We present detailed data on their performance from this implementation in this paper. To the best of our knowledge, these skew-handling algorithms are the first ones ever actually implemented in either a research prototype or a commercial parallel database system product.

A fundamental step underlying our approach is an initial pass of sampling the relations to be joined. The resulting set of samples is used in two ways: (1) they are used to predict the level of skew in the data, and hence to select the appropriate join algorithm to employ, and (2) they are used within the skew handling algorithms to determine the proper mapping of work to processors. The initial sampling in our implementation is extremely fast — approximately one percent of the time it would take hybrid hash to perform a join of the two relations assuming non-skewed data.

A further desirable property of our approach is that it can be easily implemented within the framework of existing parallel database systems. The modifications required to an existing system are minimal; it took us less than a person-month to add this skew-handling scheme to the Gamma prototype.

The remainder of this paper is organized as follows. Section 2 describes our algorithms and the techniques that they use to avoid skew. Section 3 describes the implementation of these algorithms within the Gamma parallel database machine. In Section 4 we present results from a series of experiments with the implementation of these algorithms. Section 5 describes related work on handling skew in parallel join operations including a comparison of these earlier techniques with our own. We present our conclusions in Section 6.

2 Algorithms

This section is composed of three parts: a description of the basic parallel hash join and how it is vulnerable to skew; the basic techniques we employ to handle skew; and the resulting new algorithms built using these basic techniques. While these techniques are described in the context of parallel hash joins, they are applicable to a wide range of parallel database algorithms. In fact, the fundamental problem with skew has nothing to do with joins. Skew can occur whenever hashing is used to parallelize a task. For example, the techniques we describe in this section can just as well be applied if a more traditional join algorithm such as sort merge is used at each processor.

2.1 Review of Basic Parallel Hash Join

At the highest level, the working of parallel hash join algorithms in a shared-nothing multiprocessor database system is simple. For concreteness, suppose that we are joining R and S, and that the join condition is $R.A = S.B$. Initially, both relations R and S are distributed throughout the system; if there are k processors, and the sizes of R and S (in tuples) are $|R|$ and $|S|$, then approximately $|R|/k$ tuples of R reside on disk at each processor. Similarly, each processor has about $|S|/k$ tuples of S on its disk.

To perform the join, each processor executes the following steps:

1. Every processor in parallel reads its partition of relation R from disk, applying a hash function to the join attribute of each tuple in turn. This hash function has as its range the numbers $0..k-1$; if a tuple hashes to value i, then it is sent to processor number i. The set of R tuples sent to processor i in this step will be denoted R_i.

2. Each processor i in parallel builds a memory resident hash table using the tuples sent to it during step 1. (This hash table uses a different hash function than the one used to repartition the tuples in step 1.)

3. Each processor in parallel reads its partition of S from disk, applying the same hash function used in step 1 to each tuple in turn. As in step 1, this hash function is used to map the S tuples to processors. The set of S tuples sent to processor i in this step will be denoted S_i.

4. As a processor receives an incoming S tuple s, the processor probes the hash table built in step 2 to see if s joins with any tuple of R. If so, an answer tuple is generated.

As mentioned above, this is a simplified description. For example, if not all of the R tuples received in step 2 fit in memory, some overflow handling scheme must be employed. Most commonly, the overflow processing is handled by partitioning R_i into smaller subparts, called *buckets*, such that each bucket is small enough to fit entirely within memory. A critical factor in determining the performance of the algorithm is the number of buckets needed for each of

the R_i; the larger the number of buckets, the more I/O necessary as the tuples in the overflow buckets of R_i and S_i are spooled to disk and then re-read to perform the join.

From the preceding description it should be clear that for good parallelization the number of tuples mapped to each processor should be approximately equal, or else load imbalances will result (this form of imbalance is what Walton [WDJ91] terms *redistribution skew*). These load imbalances could be the result of a poorly designed hash function. However, load imbalance due to a poor hash function can be removed by choosing a better hash function; the theoretical literature on hashing gives a number of techniques designed to find a hash function that with high probability performs well [CW79]. A more fundamental problem arises from repeated values in the join attribute. By definition, any hash function must map tuples with equal join attribute values to the same processor, so there is no way a clever hash function can avoid load imbalances that result from these repeated values.

A more subtle cause of load imbalance occurs when the number of matching tuples varies from processor to processor. This form of load imbalance results if the join selectivity for $R_i \bowtie S_i$ differs from the join selectivity for $R_j \bowtie S_j$. This type of load imbalance is called *join product skew* by Walton et al. [WDJ91].

2.2 Skew Avoidance Fundamentals

In the next five subsections we describe the techniques we apply to resolving both types of skew.

Range Partitioning

A basic approach to avoiding redistribution skew is to replace hash partitioning with range partitioning. The idea is that instead of allocating each processor a bin of a hash function, each processor is allocated a subrange of the join attribute value. The values that delineate the boundaries of these ranges need not be equally spaced in the join attribute domain; this allows the values to be chosen so as to equalize the number of tuples mapped to each subrange. For example, if the join attribute values appearing in the relation are {1,1,1,2,3,4,5,6}, and there are two processors, one could choose "3" to be the splitting value, sending tuples with values 1 and 2 to processor zero and tuples with join attribute values $3-6$ to processor one.

In general, if there are k processors, then there will be $k-1$ "splitting values" delineating the boundaries between contiguous ranges. We call these $k-1$ splitting values the "partitioning vector." The partitioning vector is "exact" if it partitions the tuples in the relation into exactly equal sized pieces. While computing an exact partitioning vector is difficult, an attractive aspect of range partitioning is that it is relatively easy to determine an *approximate* partitioning vector via sampling; that is, without examining the entire relation. This technique of sampling for approximate splitting vectors has been used previously in DBMS algorithms for evaluating non-equijoins [DNS91a] and for parallel external sorting [DNS91b]. A theoretical investigation of

R(K1,A)	S(K2,B)
(1,3)	(1,1)
(2,3)	(2,2)
(3,3)	(3,3)
(4,3)	(4,4)
	(4,5)

Table 1: Example relations R and S.

the performance of sampling-based range splitting appears in [SN92].

In a two relation join, say $R \bowtie S$, the question arises whether an algorithm should attempt to balance the number of R tuples per node, or the number of S tuples per node, or the sum of the R and S tuples per node. The answer is not always clear, but a useful general observation is that an imbalance in the number of building tuples is much worse than an imbalance in the number of probing tuples, since an imbalance in the number of building tuples per site gives rise to extra buckets in the local subjoins, driving up the number of I/Os significantly. This observation is validated by results that we reported in [SD89] and by our experimental results in Section 4.

Subset-Replicate

One complication arises with join processing via range partitioning in the presence of highly skewed data: for equal sized partitions, it might be necessary to map a single data value to multiple partitions. For example, if the join attribute values are {1, 1, 1, 1, 1, 1, 2, 3}, an equal-sized partitioning would map {1, 1, 1, 1} to processor zero and {1, 1, 2, 3} to processor one. If using a range partitioning that assigns single values to more than one partition, one must take care to ensure that all possible answer tuples are produced. A simple solution would be to send all tuples with the repeated join attribute value to all processors to which that value is mapped, but this only results in multiple processors doing exactly the same work and producing the same answer tuples at multiple sites.

It is sufficient to send *all* tuples with the repeated attribute value from one relation to all sites to which that value is mapped, and to send each tuple with the repeated attribute value in the other relation to exactly one of the sites with repeated values. We call this technique *subset-replicate*. (Subset-replicate is similar to the fragment-replicate technique proposed for distributed relational query processing by Epstein et al. [ESW78].) As an example, suppose we are joining R and S with the join predicate $R.A = S.B$. Furthermore, suppose that the relations R and S contain tuples as shown in Table 1.

Suppose we wish to join R and S on two processors. The splitting vector in this case is a single value (since there are only two processors), the value "3." Then a subset-replicate partitioning onto two processors p_0 and p_1 might send the R tuples $(1,3)$ and $(2,3)$ to processor

p_0 and the R tuples $(3,3)$ and $(4,3)$ to processor p_1. This is the "subset" part of the partitioning. Since the R tuples were subsetted, for correctness the S tuples with the join attribute value 3 must be replicated among both processors. This means that the S tuples $(1,1)$, $(2,2)$, and $(3,3)$ will be sent to p_0, while the S tuples $(3,3)$, $(4,4)$, and $(4,5)$ will be sent to p_1.

Again the question arises whether to replicate the building (inner) relation and to subset the probing (outer) relation or vice-versa. While there are clearly situations where either will out perform the other, again a reasonable heuristic is to subset the building relation and replicate the probing relation. The motivation for this heuristic is that it is critical that the portion of the building relation mapped to each processor be as small as possible so as to minimize the number of buckets in the join.

Weighting

Another complication that arises with range partitioning is that it will often be the case that a join attribute value appears a different number of times in different partitions. For example, suppose that the join attribute values in a 12 tuple relation are {1,2,3,4,4,4,4,4,4,4,4,6}, and that we wish to partition over three processors p_0, p_1, and p_2. Then an even partitioning vector would be $[4,4]$, meaning that tuples with the join attribute value 4 should be mapped to all three processors. Since a total of 8 tuples have the join attribute value "4", to balance the load evenly among the 3 processors, 1/8 of the tuples with 4 as the join attribute must be directed to processor p_0 (along with join attribute values 1, 2, and 3), 1/2 to processor p_1, and 3/8 to processor p_2 (along with join attribute value 6).

We refer to this technique for distributing replicated values for the subsetted relation as *weighted range partitioning.*

Virtual Processor Partitioning

This and the next subsection deal with the problem of join product skew. For concreteness, suppose that we are joining two 10,000 tuple relations and that in each relation the join attribute value "1" appears 1,000 times and no other join attribute value appears more than once. Also, assume that we have 10 processors. Then if we use equal sized range partitioning, all 1000 tuples with "1" as their join attribute value from both relations will be mapped to processor zero, meaning that processor zero will be asked to generate 1,000,000 result tuples. There is no way to remedy this problem by choosing a set of 9 splitting values; too many 1's will be mapped to some processor in every case.

The solution to this problem is to choose many more partitions than there are processors. This idea has appeared many times before in the skew join literature with respect to hash bucket partitioning; the first reference to the technique is probably in [KTMo83]. We refer to the technique of using multiple range partitions per node as *virtual processor partitioning.* In the previous example, if we chose 100 buckets per processor, for a total of 1000 buckets,

we would have a fine enough granularity to resolve this problem. In particular, the 1000 1's would be spread among 100 buckets (subranges), each of which could be mapped to a different processor. This of course leaves open the question of how these virtual processor partitions are to be mapped to the actual processors. We considered two techniques for this, both described in the next subsection.

Load Scheduling

We consider two basic techniques for mapping virtual processor partitions to actual processors:

1. Round robin.

   This is the simplest scheme — if there are k processors, the ith virtual processor partition is mapped to actual processor $i \bmod k$.

2. Processor scheduling.

   In this scheme, for each virtual processor partition i, we compute an estimate of the cost c_i of joining the tuples of R_i and S_i. Any formula for estimating the cost of a join could be used; we chose the simple technique of estimating that

   $$c_i = |R_i|_{\text{est}} + |S_i|_{\text{est}} + |R_i \bowtie S_i|_{\text{est}}$$

   where $|R_i|_{\text{est}}$ is an estimate of the number of R tuples mapped to partition i, $|S_i|_{\text{est}}$ is an estimate of the number of S tuples mapped to partition i, and $|R_i \bowtie S_i|_{\text{est}}$ is an estimate of the number of tuples in $R_i \bowtie S_i$. We computed this estimate of the size of $R_i \bowtie S_i$ by assuming that the join attribute values in each of R_i and S_i were uniformly distributed between the endpoints of the range for virtual processor partition i. Once this estimate for the cost of the joining of the virtual processor partitions has been computed, any task scheduling algorithm can be used to try to equalize the times required by the virtual processor partitions allocated to the physical processors. We used the heuristic scheduling algorithm known as "LPT" [Gra69].

   This approach is similar to that used by Wolf et al. [WDYT90] in scheduling hash partitions, although in that paper the statistics used to schedule these partitions are gained by a complete scan of both relations rather than by sampling, and hash partitioning is used instead of range partitioning.

2.3 Algorithm Description

The algorithms that we implemented can be described in terms of the skew handling techniques defined above. But first we need to discuss how the approximate splitting vectors are computed. For each algorithm except hybrid hash, we first used sampling to compute a statistical profile of the join attribute values of the two relations to be joined. We obtained this sample by using

stratified sampling [Coc77] with each stratum consisting of the set of tuples initially residing at a processor. Within each processor, the sampling was performed using page-level *extent map sampling*. Extent map sampling is described in Section 3. Issues involving stratified sampling and page level sampling are discussed in [SN92]. We now describe the skew handling algorithms.

1. **Hybrid hash.**

   This is just the basic parallel hybrid hash algorithm (with no modifications for skew handling.) A description of this algorithm and some alternatives appears in [SD89].

2. **Simple range partitioning.**

   At the top level, this algorithm works as follows:

   (a) Sample the building (inner) relation.

   (b) Use the samples to compute an approximate partitioning vector. The number of partitions defined by the partitioning vector is equal to the number of processors.

   (c) Redistribute the building relation using the approximate partitioning vector to determine to which processor the tuples should go.

   (d) Build an in-memory hash table containing as many building relation tuples as possible. Overflow tuples are partitioned into buckets sized so that each such bucket will fit in main memory [SD89].

   (e) Redistribute the probing (outer) relation using the same approximate partitioning vector as in step 3.

   (f) For each tuple of the probing relation probe the in-memory hash table, outputting a join result tuple for each match. If overflow occurred in step 4, probing tuples corresponding to one of the overflow buckets of the building relation are written directly to disk. Once, all the probing tuples have been received, the overflow buckets of the building and probing relations are processed.

3. **Weighted range partitioning.**

   This algorithm is the same as range partitioning except that instead of simple range partitioning, tuples are redistributed using weighted range partitioning.

4. **Virtual processor partitioning - round robin.**

   This algorithm is the same as range partitioning except that instead of having the number of partitions equal the number of processors, the number of partitions is a multiple of the number of processors. The exact number of partitions is a parameter of the algorithm. The partitions are allocated to processors using round robin allocation.

5. **Virtual processor partitioning - processor scheduling.**

This algorithm is the same as virtual processor partitioning - round robin except that instead of using round robin allocation of partitions to processors, processor scheduling using LPT is used.

3 Implementation Details

In this section we describe some of the details of the implementation of the skew handling algorithms within Gamma. We begin by explaining how we sampled the relations, and then consider the modifications to Gamma that were necessary for the remainder of the algorithms.

Sampling Implementation

As mentioned in Section 2, we use stratified sampling to obtain a sample from relations distributed throughout the multiprocessor. In stratified sampling, if a k node multiprocessor needs to take n samples, each processor takes n/k samples from its local partition of the database. Although this is not a simple random sample of the entire relation, a stratified sample is sufficient for our purposes.

Stratified sampling requires that each processor take some specified number of samples from its partition of the database. A number of techniques have been proposed for this problem, notably sampling from B^+ trees [OR89], sampling from hash tables [ORX90], and using a dense index on a primary key [DNS91a]. In this section we describe a new technique that we call *extent map sampling.*

Extent-based sampling requires neither an index on a dense primary key nor an index on any other attribute. Our scheme hinges on the fact that many systems allocate pages in contiguous units called *extents*, and record information about where the pages of a file are stored by linking together the extents for the pages of the file. This information is maintained in a small memory-resident data structure. Moreover, the address of a page within an extent can be found by adding an offset to the address of the first page of this extent. Given this information, we can select a random page or tuple as follows: generate a random number r between one and the number of pages in the file (relation). Find the address of the rth page of the file by chaining down the linked list of extents. If a random page is desired, then this page can be brought in; if a random tuple is desired, we follow this I/O by randomly choosing one of the tuples in the page.

The above correctly chooses a random page if the pages in the relation have the same number of tuples. However, if they do not we will need acceptance/rejection sampling to accept or reject a randomly chosen page so that the inclusion probabilities for each tuple of the relation is identical. If all pages have the same number of tuples then we require exactly one I/O to fetch a random tuple. If they do not, then the average number of I/O's required for fetching a random tuple is the inverse of the fill-factor. Therefore, if the fill-factor is more than 50% we would need at most two I/O's on an average to fetch a random tuple. This is

still better than the previous index-based methods even assuming that the previous methods have no wasted I/O's due to acceptance rejectance sampling. For this reason we have adopted extent-map sampling in our implementation.

We also used page-level sampling in our implementation. This means that after a random page has been selected and read into memory (using extent map sampling), we add every tuple on that page to the sample. This in effect boosts the number of samples per I/O by a factor equal to the average number of tuples per page. This technique is most efficient if the correlation on the join attribute within a page is low.

Implementation in Gamma

In order to investigate the performance of our skew handling algorithms, we implemented the algorithms using Gamma [DGS+90] 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.

Gamma is built on top of an operating system designed specifically for supporting database management systems. NOSE provides multiple, lightweight processes with shared memory. A non-preemptive scheduling policy is used to help prevent convoys [BGMP79] from occurring. NOSE provides communications between NOSE processes using the reliable message passing hardware of the Intel iPSC/2 hypercube. File services in NOSE are based on the Wisconsin Storage System (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.

Before beginning this work, Gamma already contained the code needed to perform a parallel hybrid hash join. The critical code that needed to be added to the system in order to incorporate our new skew handling join algorithms were

1. code to do the parallel stratified page level extent map sampling,

2. code to sort the resulting samples and build the required approximate splitting vectors, and

3. code to redistributes tuples using the new distribution types (e.g., subset-replicate) required by our algorithms.

Items 1 and 2 above were straightforward. We now discuss the changes to the redistribution code in more detail.

Basic parallel hybrid hashing in Gamma makes use of a data structure called a *split table* [DGS+90, DG92]. This data structure contains entries that are (hash bucket, processor number) pairs. If k processors are being used to execute a relational operation, then the split tables have k entries. The semantics are such that any tuple that hashes to a given hash bucket should be sent to the processor number in the split table entry for that hash bucket. Each processor executing an operation has a copy of this split table. In a given processor, associated with the split table are k outgoing buffer pages, one for each processor. When a tuple maps to a given hash bucket, it is added to the corresponding buffer page; when this page fills, a message containing this page is sent to the target processor.

To add basic range partitioning, we added a new type of split table called a *range* split table. This was a simple modification; the only change is that entries of the split table correspond to ranges of join attribute values instead of corresponding to hash buckets. When deciding where to send a tuple, instead of hashing the join attribute value to find the corresponding entry, the range split table is searched to find the range containing the join attribute value. If a tuple t maps to more than one range (e.g., if there are repeated values in the split table), then, during redistribution of the building (inner) relation, one of the duplicate ranges is selected at random and t is sent to the corresponding processor. During redistribution of the probing (outer) relation, t is sent to the processors corresponding to all of the containing subranges.

To add weighted range partitioning, we augmented the basic range split table to contain weights for the upper and lower boundary values of each range in the table. These weights are computed from the sorted set of samples at the time when the partitioning values are being computed. Then, during the redistribution of the building relation, instead of sending tuple t to a randomly selected subrange, a subrange is selected with a probability that reflects the weights in the weighted-range split table.

The most obvious way to add virtual processor range partitioning would be to expand these basic range splitting tables to add more entries than processors. The difficulty in doing so is that the lower level Gamma code assumes that there will be exactly one outgoing buffer page for every entry in the split table. For large numbers of virtual processors, the space required by this scheme is prohibitive. For example, for 30 processors and 50 virtual processor ranges per processor it would require 1500 output buffers (12 megabytes with 8K byte network packets) per node. This is more than the total amount of memory per node in our sytem.

To solve this problem we used a two-level split table. The upper level table contains the same number of entries as the number of virtual processor partitions. The lower level table contains one entry per processor. Each entry in the upper table consists of a (range, lower split table entry number) pair. When a tuple is being processed to decide to which processor

it should be sent, first a lookup is performed on the upper table to determine the set of virtual processor ranges in which the join attribute value of the tuple appears. Next the entries for these ranges are examined to determine to which lower level entries the tuple belongs. From this set of entries in the lower level table the system can determine to which processors the tuple should be sent. Only one buffer page per destination processor is used.

4 Experiments and Results

Test Data

For the purposes of this experiment we wanted to use a set of test data that was simple and intuitively easy to understand, yet that would stress all of our skew handling algorithms. One option would have been to generate relations with attributes drawn from standard statistical distributions (like Zipf and normal.) We decided against this because we found that relations with such attributes make the experiments much harder to understand and control. For example, suppose we wish to perform a set of joins on a pair of relations, varying the level of skew in both relations, yet keeping the answer size approximately constant? This is difficult to do with sets of Zipfian distributions.

To remedy this problem we generated relations with a number of integer attributes, each with various amounts of "scalar skew" — that is, in an N tuple relation, in each attribute the constant "1" appears in some fixed number of tuples, while the remaining tuples contain values uniformly distributed between two and N. The use of such a distribution has three major benefits. First, it makes it easy to understand exactly what experiment is being performed. Second, it is easy to keep the answer size constant over varying amounts of skew. Finally, it captures the essence of the Zipfian distribution (a small number of highly skewed values with the bulk of the values appearing very infrequently) without suffering its drawbacks. The term "scalar skew" is due to Walton et al. [WDJ91]. This is also the model of skew used by Omiecinski [Omi91].

The exact description of the attributes are as follows. In each case, we are assuming a relation of N tuples, and that $N \geq 100,000$. The attributes relevant to our experiments are x1, x10, x100, x1000, x10000, x20000, x30000, x40000, and x50000. The number following the "x" in each case is the number of tuples in which the value "1" appears in the join attribute (these tuples are chosen at random). The remainder of the tuples have a join attribute value chosen randomly from 2 to N, where N is the number of tuples in the relation. For example, the x10 attribute has the semantics that the value "1" appears in exactly ten randomly chosen tuples. The remaining $N - 10$ tuples contain values uniformly chosen at random between 2 and N. The rationale for choosing these attributes should become more apparent in the following set of experiments. In addition to the attributes listed above, each tuple contained a string attribute to pad the length of each tuple to 100 bytes. In all of our experiments below we used relations of 500,000 tuples. Thus, each relation occupies approximately 50 megabytes of disk space.

All experiments were conducted using 30 processors with disks. Speedup or scaleup experiments were not performed as we were more interested in focusing on the relative performance of the different algorithms. Furthermore, previous join [DGG+86, DGS+90, DGS88, DNS91a, SD89] and sorting [DNS91b] tests demonstrated that the Gamma provides linear speedup and scaleup over a wide range of different hardware and software configurations.

Single Skew Experiments

In the first set of experiments we ran the building relation was skewed and the probing relation was uniform. This models a very common sort of join in practice — joins between a key of one relation and the corresponding foreign key in another. Each data point is the average of 5 experiments. For the range, weighted range, and virtual processor range partition round robin the number of samples on the building relation was fixed at 14,400 (the probing relation is not sampled in these algorithms.) For the virtual range partition processor scheduling algorithm, we took 14,400 samples of both the building and probing relations. For the virtual processor range partitioning algorithms we use 60 virtual processors per processor. The results of the experiment appear in Table 2.

Alg.	x1 $\bowtie$ x1	x10K $\bowtie$ x1	x20K $\bowtie$ x1	x30K $\bowtie$ x1	x40K $\bowtie$ x1	x50K $\bowtie$ x1
HH	33.0	52.2	79.5	DNF	DNF	DNF
Range	43.1	43.4	58.9	DNF	DNF	DNF
W. Range	41.8	41.9	51.7	52.2	52.9	52.6
VP-RR	43.9	44.2	44.0	43.4	43.8	43.3
VP-PS	47.7	47.3	47.5	47.6	47.9	47.6

Table 2: Effect of skewed building relation.

In Table 2, entries marked "DNF" means that the algorithm did not finish. The reason these tests did not finish was that in those cases marked "DNF", the algorithms mapped more tuples with "1"s in the join attribute to a single processor than can simultaneously fit in the memory of that processor. In the current Gamma implementation, the per-node hybrid hash code does not handle this extreme case. We see that Hybrid Hash (HH) is clearly the algorithm of choice for the zero skew case (x1 $\bowtie$ x1). This is because when compared to the skew handling algorithms, (1) Hybrid Hash does not incur the overhead of collecting the samples, sorting the samples, and computing an approximate splitting vector, and (2) in Hybrid Hash, to determine a destination processor during redistribution one need only compute a hash function, while in all the other algorithms it is necessary to search a sorted list for the appropriate range entry.

The difference in performance for Range Partitioning (Range) and Weighted Range Partitioning (W. Range) at zero skew is an artifact of the implementation — Weighted Range Partitioning was implemented second and uses a more efficient table search during repartition-

ing. We expect that if Range Partitioning were reimplemented using this new code, it would be slightly faster at zero skew since it doesn't need to check the weights before choosing a destination in the subset phase.

At x10K, both Range Partitioning and Weighted Range Partitioning effect the same partitioning, sending the tuples with 1's in the join attribute along with about 6K other tuples to processor zero. However, at x20K, Range Partitioning sends all 20K tuples with 1's to processor zero, while Weighted Range Partitioning sends about 16K of these tuples to processor zero and 4K of these tuples (plus about 12K other tuples) to processor one. Weighted Range Partitioning performs worse on x10K than on x1 because even though the same number of tuples are distributed to each processor in both cases, in the x10K case the join hash table for processor zero contains one bucket with 10K tuples (the bucket to which "1" is mapped.) At 20K the situation is even worse, as there is a bucket with about 16K ones in that case.

Virtual Processor Range Partitioning with Round Robin allocation (VP-RR) starts off at zero skew with slightly higher overhead than Weighted Range because during redistribution, to determine a destination processor it must search a much bigger range table (bigger by a factor of 60.) Virtual Processor Range Partitioning with Processor Scheduling (VP-PS) has even more overhead, since it must sample and sort the probing relation and then run the LPT scheduling algorithm. However, in the skewed cases both these algorithms outperform Range and W. Range because they map the tuples with 1's to more processors, avoiding the large hash table entry effect.

Next we wanted to test the effect that a skewed probing relation would have on the algorithms. Note that since the first four algorithms do not sample the probing relation, these algorithms use the same splitting vector independent of the skew in the probing relation. For this reason, the performance deteriorates rapidly, so we do not go beyond x1 $\bowtie$ x20K. Note that Hybrid Hash does relatively well here. VP-PS samples the probing relation, but its estimates of the per virtual processor execution times were too inaccurate to provide good performance.

Algorithm	x1 $\bowtie$ x1	x1 $\bowtie$ x10K	x1 $\bowtie$ x20K
HH	33.0	44.5	55.3
Range	43.1	53.3	63.7
W. Range	41.8	51.0	61.7
VP-RR	43.9	52.2	63.0
VP-PS	47.7	58.1	67.9

Table 3: Effect of skewed probing relation.

An alternative approach to handling single relation skew would be to sample the probing relation, then use these samples to compute a splitting vector that could be used for both the building and probing relations. We did not pursue this approach for the following reason: if the probing relation is highly skewed, and we distribute the building relation using a splitting

vector that evenly distributes the probing relation, then greatly varying numbers of building tuples are sent to each processor. This in turn causes some processor(s) to use many more buckets that would be necessary if the building relation were evenly distributed, which will cause performance to suffer.

Join Product Skew

In this subsection we present experiments in which both relations that participate in the join are skewed. In general, this sort of skew is much harder to deal with than skew in a single relation. Intuitively, the problem is that in join product skew, a relatively small number of repeats can cause a tremendous blowup in the number of tuples generated in the join. For example, if we join the two relations using the join clause x10000 $\bowtie$ x10000, the result will have 10^8 tuples generated due to matches of tuples with ones in the join attributes. This result would be 20G bytes. In addition to exceeding the capacity of our disk drives, we don't think such queries make any sense. Accordingly, we decided to experiment with more modest skews. The first set of experiments below shows the performance of the algorithms using the same configuration (number of samples, number of virtual processors per node) as in Table 2.

Algorithm	x10K $\bowtie$ x10	x1K $\bowtie$ x100	x100 $\bowtie$ x1000
HH	143.6	144.0	144.0
W. Range	148.2	148.2	149.4
VP-RR	49.7	85.8	151.0
VP-PS	56.3	94.1	155.0

Table 4: Performance on data with join product skew.

The joins in Table 4 were designed so that the result size is roughly comparable to that in Tables 2 and 3. In each case the result contains about 600K tuples, 100K of which are due to joining tuples that contain ones in the join attribute. It is clear that only the virtual processor algorithms have significant success in dealing with this sort of skew. Intuitively, the reason is that in each of the Range and Weighted Range algorithms, the skew in the relation is not enough to cause tuples with one's in the join attribute to be sent to more than one processor.

With the exception of the x100 $\bowtie$ x1000 join, both of the virtual processor algorithms have enough virtual buckets that the one's are mapped to enough processors to distribute the work. For the x100 $\bowtie$ x1000 join, the round robin algorithm fails to distribute the one's because there are so few in the building relation. The virtual processor range partitioning processor scheduling algorithm also fails to distribute the one's into multiple buckets, again because its estimates of the work required per virtual processor are too inaccurate.

It is clear that the performance of the virtual processor range partition algorithms is critically dependent upon the number of virtual processors per processor. Table 5 explores the

performance of the round robin variant on the join x10000 $\bowtie$ x100 for various numbers of processor per node. (Since in our experiments the processor scheduling variant was uniformly worse than the round robin variant, we omit the data points for that algorithm.) The table shows the clear trend that the more virtual processors, the better the performance. The reason for this is that the tuples with "1"s are being distributed over more and more (actual) processors, achieving better load balancing.

virt. procs.	1	5	10	20	30	60
exec. sec.	147.2	95.3	64.0	54.0	51.8	49.7

Table 5: Dependence on number of virtual processors, x10000 $\bowtie$ x100, virtual processor range partitioning.

Finally, we wanted to illustrate the dependence of virtual processor range partitioning on the number of samples. Table 6 lists the average time as a function of the number of samples for the virtual processor range partition round robin algorithm as a function of the number of samples for the join x10000 $\bowtie$ x100. Again, since virtual processor range partitioning with round robin allocation was uniformly the best skew handling algorithm, we only present data for it. Note that the performance is relatively stable independent of the number of samples. The general trend is that taking too few samples results in poor load balancing, while taking too many samples results in too much overhead due to sampling (notice in Table 6 that the overall running times dip from 1800 to 3600 samples and then begin to rise again.)

number of samples	1800	3600	7200	14400
execution time (sec)	49.0	47.8	49.0	49.7

Table 6: Dependence on number of samples, x10000 $\bowtie$ x10, virtual processor range partitioning.

Finally, we would like emphasize that the virtual processor range partition round robin is exceedingly successful at balancing the load among the processors during the execution. Table 7 gives maximum and minimum times (over all processors) to complete the building phase (that is, redistributing the building relation and building an in-memory hash table) and the entire join of x1000 $\bowtie$ x10. As before, we used 14400 samples and 60 virtual processors per processor. Note that the total time (49.77 seconds) differs from the time reported in for this join in Table 4. This is because the times presented in that table are averages over five runs, whereas the times in Table 7 are from a single run. The difference between the maximum and minimum times for the building phase is less than 6%; the difference for the total execution time is about 2%.

Phase	min seconds	max seconds
Building	15.55	16.48
Complete Join	48.72	49.77

Table 7: Maximum and minimum times over all processors, x10000 $\bowtie$ x10, virtual processor range partitioning.

5 Related Work

There has been a wealth of research in the area of parallel join algorithms. Originally, join attribute values were assumed to be uniformly distributed and hence skew was not a problem (see, for example, [BFKS87, Bra87, DG85, DGS88, KTMo83].) As parallel join algorithms have matured, this uniformity assumption has been challenged (see, eg., [LY90, SD89]). In this section, we examine a number of previously proposed algorithms for dealing with data skew and compare these algorithms with our own.

5.1 Walton, Dale, and Jenevein

Walton et al. [WDJ91] present a taxonomy of skew in parallel databases. First, they distinguish between *attribute value skew (AVS)* which is skew inherent in the dataset, and *partition skew* which occurs in parallel machines when the load is not balanced between the nodes. AVS typically leads to partition skew but other factors are also involved. These include:

1. Tuple Placement Skew (TPS): The initial distribution of tuples may vary between the nodes.

2. Selectivity Skew (SS): The selectivity of selection predicates may vary between nodes, for example, in the case of a range selection on a range-partitioned attribute.

3. Redistribution Skew (RS): Nodes may receive different numbers of tuples when they are redistributed in preparation for the actual join.

4. Join Product Skew (JPS): The join selectivity on individual nodes may differ, leading to an imbalance in the number of output tuples produced.

Walton et al. use an analytical model in order to compare the scheduling hash-join algorithm of [WDYT90] and the hybrid hash-join algorithm of Gamma [SD89, DGS+90]. The main result is that scheduling hash effectively handles RS while hybrid hash degrades and eventually becomes worse than scheduling hash as RS increases. However, unless the join is significantly skewed, the absolute performance of hybrid hash is significantly better than that of scheduling hash.

5.2 Schneider and DeWitt

In [SD89], we explored the effect of skewed data distributions on four parallel join algorithms in an 8 processor version of the Gamma database machine. The experiments were designed such that TPS and SS were absent. For the tested AVS (normally distributed values), the hash function used in the redistribution phase was quite effective in balancing the load and hence RS was low. Likewise, JPS was low.

The overall results were that the parallel hash-based join algorithms (Hybrid, Grace, and Simple) are more sensitive to RS resulting from AVS in the "building" relation (due to hash table overflow) but are relatively insensitive to RS for the "probing" relation. Experiments with "double-skew" (which lead to JPS) were not run but we extrapolated that the problems would be worse because this case is a superset of the RS for the building relation.

5.3 Kitsuregawa and Ogawa

Kitsuregawa and Ogawa [KO90] describe two algorithms, *bucket-converging parallel hash-join* and *bucket-spreading parallel hash join*. The bucket-converging hash join is a basic parallelization of the GRACE join algorithm [KTMo83]. Relation R is read from disk in parallel and partitioned into p buckets (where p is much larger than k, the number of nodes). Since each bucket is statically assigned to a particular node, all of R is redistributed during this phase of the algorithm. Next, the size of each bucket is examined, and, if necessary, enough buckets are redistributed so that the sum of the sizes of the buckets at each processor is balanced. Relation S is processed similarly. In the last phase, all of the respective buckets of R and S on each node are joined locally.

As they point out, the first phase of this algorithm (the initial repartitioning) is very susceptible to RS. As an alternative, they propose a bucket-spreading hash join algorithm. In this algorithm, relations R and S are partitioned into p buckets as before but each bucket is horizontally partitioned across all available processors during the initial repartitioning phase. During the second phase of the algorithm, a very sophisticated network, the Omega network, is used to redistribute buckets onto the nodes for the local join operation. The Omega network contains logic to balance the load during the bucket redistribution.

Simulation results are presented for the two algorithms where AVS is modeled using a Zipfian distribution. When the data is uniformly distributed, the two algorithms are almost identical. The bucket-spreading algorithm is shown to effectively reduce RS in the presence of increasing AVS, while the bucket-converging algorithm suffers.

When compared to our weighted-range and virtual processor algorithms, both of these algorithms are likely to have higher response times. In particular, our algorithms redistribute both the joining relations exactly once. Their bucket-spreading algorithm redistributes both relations twice. In addition, if the two relations do not fit in memory, an extra write and read of both relations to disk will be required between the two repartitioning phases. The bucket-

converging algorithm, on the other hand, incurs extra redistribution and I/O costs only for those buckets that must be redistributed in order to balance the load among the processors. However, as they point out, this algorithm is very susceptible to RS.

5.4 Hua and Lee

Hua and Lee [HL91] proposed three algorithms for processing parallel joins in the presence of AVS. The first algorithm, *tuple interleaving parallel hash join*, is based on the bucket-spreading hash join algorithm of Kitsuregawa and Ogawa [KO90]. The major difference is that instead of relying on a specially designed intelligent network for mapping buckets to nodes, this decision is handled in software by a coordinator node.

The second algorithm, *Adaptive Load Balancing parallel hash join*, tries to avoid much of the massive data redistribution incurred by the tuple interleaving algorithm. In the case of mild skew, a more selective redistribution is likely to perform better. In this algorithm, relations R and S are partitioned into p buckets where each bucket is statically assigned to a single node. Instead of immediately performing local joins, though, a *partition tuning* phase is executed in which a best-fit decreasing heuristic is used to determine which buckets to retain locally versus which ones to redistribute. This algorithm is basically identical to Kitsuregawa and Ogawa's bucket-converging algorithm,

The final algorithm, *Extended Adaptive Load Balancing parallel hash join*, is designed for the case of severe skew. Relations R and S are partitioned into p buckets where each bucket is stored locally. Next, all nodes report the size of each local bucket to the coordinator who decides on the allocation of buckets to nodes. The allocation decision is broadcast to all the nodes and all the buckets are redistributed across the network. Local joins of respective buckets are then performed on each node. The basic form of this algorithm is identical to that of Wolf et al. [WDYT90]. The algorithms differ in the computation of the allocation strategy.

The three algorithms are compared using an analytical model. The basic results are that the tuple interleaved and extended adaptive load balancing algorithm are unaffected by skew in the size of partitions while the performance of the adaptive load balancing algorithm and the bucket-converging algorithm eventually cross over and become much worse as the skew increases.

Since the first two algorithms are basically identical to those of Kitsuregawa, they have the same relative performance to our algorithms. Like our algorithms, the extended adaptive load balancing parallel hash join algorithm repartitions each relation exactly once. However, unless both relations fit in memory, an extra read and write of both relations occurs during the initial bucket forming phase. The cost of this step is certainly higher than the cost we incur sampling one or both relations (about 1/2 second each in our implementation).

5.5 Wolf, Dias and Yu

Wolf et al. [WDYT90], propose an algorithm for parallelizing hash joins in the presence of severe data skew. The scheduling hash algorithm is as follows. Relations R and S are read, local selections or projections are applied, and the results are written back locally as a set of coarse hash buckets. Additionally, statistics based on a finer hash function are maintained for each bucket. Next, a scheduling phase occurs in which a coordinator collects all the fine and coarse bucket statistics and computes an allocation of buckets to nodes. The allocation strategy is broadcast to all nodes and relations R and S are redistributed across the network accordingly. Hash-joins are then performed locally for each bucket.

Several heuristics are proposed for computing the allocation strategy in the scheduling phase including *longest processing time first*, *first fit decreasing*, and *skew*.

An analytical model is used to briefly compare the strategies. AVS is modeled with a zipfian distribution. No TPS or SS skew occurs. A double-skew (skew in both join relations) style join is specifically modeled. The load-balancing heuristics are shown to be highly effective in balancing the load especially as the number of processors becomes large. However, no comparison is made with the performance of other join algorithms (skew handling or non-skew handling.)

Like Hua's extended adaptive load balancing parallel hash join algorithm, this algorithm incurs an extra read and write of both relations during the initial bucket forming phase. The cost of this step will certainly be higher than the cost of sampling both relations. However, it may be the case that the increased accuracy in skew information that is obtained by looking at every tuple will sufficiently improve the variance in the response time among the processors that the cost of the extra read and write pass is worthwhile. Without implementing both algorithms on the same hardware and software base it is probably impossible to determine precisely which algorithm provides the best overall performance.

5.6 Omiecinski

Omiecinski [Omi91] proposed a load balancing hash-join algorithm for a shared memory multiprocessor. The algorithm is based on the bucket-spreading algorithm of Kitsuregawa and Ogawa [KO90]. It differs in that it doesn't rely on special-purpose hardware, it assigns buckets to processor(s) using a first-fit decreasing heuristic, and it has other optimizations for the shared-memory environment.

Analytical and limited experimental results from a 10 processor Sequent machine show that the algorithm is effective in limiting the effects of AVS even for double-skew joins. (AVS is modeled by having a single value account for X% of the relation while the other 1-X% of the values are uniformly distributed.)

6 Conclusion

The algorithms for skew handling proposed in this paper represent a simple way to augment existing parallel database systems to make their performance more robust in the presence of skewed joins. The modifications needed to install these changes in an existing system are simple — all that is needed is to add extent-map sampling (or some equivalent), support for subset-replicate virtual processor split tables, and finally a small amount of code to analyze the samples and build the necessary split tables.

The experiments we performed suggest the following approach to running multiprocessor joins:

1. Take a pilot sample of both relations involved in the join.
2. Inspect the resulting set of samples to determine which relation is more highly skewed (by counting the number of repeated samples in each.)
3. If neither of the relations appears skewed, revert to simple hybrid hash.
4. If at least one of the relations appears to be skewed, use the virtual processor range partition round robin join algorithm. The most skewed relation should be the building relation.

This scheme incorporates a number of heuristics, and, like all optimizer heuristics, it can be tricked into choosing a sub-optimal plan in some situations. Yet it is simple, implementable, and in general runs non-skewed joins in time comparable to that of standard hybrid hash (the overhead outlined above takes just a few seconds in our implementation) and runs skewed joins without suffering the terrible worst-case performance that would result from running hybrid hash on highly skewed data.

A number of interesting open questions remain to be addressed in future work. First, as our experiments illustrate, the virtual processor range partitioning algorithm depends critically on the number of virtual processors chosen. The optimal number for this parameter depends upon the system configuration (most importantly the number of processors) and how little skew you are willing to tolerate. The values we used in our experiments (60 virtual processors per processor) are reasonable and performed well over the test data, but we do not claim that they are globally optimal.

Second, in this work we did not address the question of how to handle joins in which the operands are of greatly different size. Our experience from these experiments suggest that a critical point is to keep the number of buckets of the building relation to a minimum. There are two ways that a large number of buckets could result: a large building relation, or a skewed building relation. A reasonable heuristic is that if the relations are of roughly comparable size, the more skewed relation should be the building relation; if they are of very different size, then the smaller relation should be the building relation and skew should be handled by building

a split table based upon samples of the probing relation. We intend to experiment with this heuristic in future work.

Finally, as the number of processors in the system grows to the thousands, the overhead of sorting and analyzing the samples will grow (the cost of obtaining the samples does not, as we can use a constant number of samples per processor as the system scales.) It is not clear that this overhead will grow as fast as the cost of performing the join itself (if one is using 1000 processors for a join, presumably it is a big join!), but still there is room for reducing this overhead by doing some of the processing in parallel instead of doing everything at a central coordinating processor. For example, as a first step every processor could sort its local set of samples before sending them to the coordinator, which could then do a simple merge instead of a sort.

7 Acknowledgments

This research was supported by donations from DEC, IBM (through an IBM Research Initiation Grant), NCR, and Tandem. Without their generous support, this research would not have been possible.

References

[BFKS87] C. Baru, O. Frieder, D. Kandlur, and M. Segal. Join on a cube: Analysis, simulation, and implementation. In M. Kitsuregawa and H. Tanaka, editors, *Database Machines and Knowledge Base Machines*. Kluwer Academic Publishers, 1987.

[BGMP79] M. W. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *Operating System Review*, 13(2), 1979.

[Bra87] Kjell Bratbergsengen. Algebra operations on a parallel computer – performance evaluation. In M. Kitsuregawa and H. Tanaka, editors, *Database Machines and Knowledge Base Machines*. Kluwer Academic Publishers, 1987.

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

[Coc77] William G. Cochran. *Sampling Techniques*. John Wiley and Sons, Inc., New York, New York, 3 edition, 1977.

[CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[DG85] David M. DeWitt and Robert Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the Twelfth International Conference on Very Large Databases*, pages 151–164, Stockholm, Sweden, 1985.

[DG92] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 1992. To appear.

[DGG+86] David J. Dewitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA — a high performance dataflow database machine. In *Proceedings of the Twelfth International Conference on Very Large Databases*, pages 228–237, Kyoto, Japan, August 1986.

[DGS88] David J. DeWitt, Shahram Ghandeharizadeh, and Donovan Schneider. A performance analysis of the GAMMA database machine. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 350–360, Chicago, Illinois, May 1988.

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

[DNS91a] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. A comparison of non-equijoin algorithms. In *Proceedings of the Eighteenth International Conference on Very Large Databases*, Barcelona, Spain, August 1991.

[DNS91b] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel external sorting using probabilistic splitting. In *PDIS*, Miami Beach, Florida, December 1991.

[ESW78] Robert Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1978.

[Gra69] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Computing*, 17:416 – 429, 1969.

[HL91] Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 525–535, Barcelona, Spain, August 1991.

[KO90] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the Super Database Computer (SDC). In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, Brisbane, England, August 1990.

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

[LY90] M. Seetha Lakshmi and Philip S. Yu. Effectiveness of parallel joins. *IEEE Transactions on Knowledge and Data Engineering*, 2(4), December 1990.

[Omi91] Edward Omiecinski. Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[OR89] Frank Olken and Doron Rotem. Random sampling from B^+-trees. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 269–278, Amsterdam, The Netherlands, August 1989.

[ORX90] Frank Olken, Doron Rotem, and Ping Xu. Random sampling from hash files. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 375–386, Atlantic City, New Jersey, May 1990.

[SD89] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 110–121, Portland, Oregon, June 1989.

[SN92] S. Seshadri and Jeffrey F. Naughton. Sampling issues in parallel database systems. In *Proceedings of the EDBT Conference*, Vienna, Austria, March 1992.

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

[WDJ91] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[WDYT90] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. IBM T. J. Watson Research Center Tech Report RC 15510, 1990.